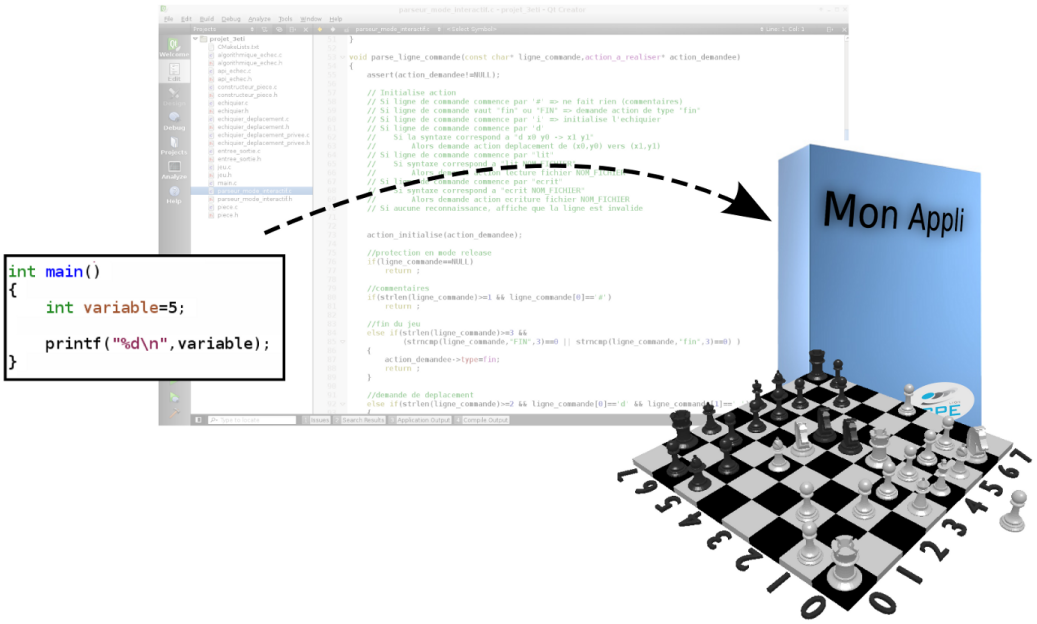
3ETI

2012/2013

Developpement logiciel en C

Fascicule de projet



3ETI, Développement logiciel en C

Présentation du projet:

Le but du projet est de réaliser le fonctionnement d'un jeu d’échec valide.
Plus spécifiquement, il consiste à implémenter l'organisation générale du jeu, et le suivi des règles du mouvement des pièces.

Le projet mettra en avant un ensemble de bonnes pratiques de codage permettant de simplifier la lecture du code, son évolutivité et sa robustesse.
Des aspects généraux de l'informatique seront également mises en pratiques (compilation, utilisation de librairies).

Organisation du code source:

Un certain nombre de fichiers sources vous est fourni. Le projet est lui-même formé de parties comportant deux exécutables.

- Le répertoire **visualiseur**:
  Permet de visualiser l'état d'un échiquier. Il contient un programme indépendant de votre projet qui permet de visualiser le placement des pièces sur l'échiquier.

  Notez que ce programme ne contient aucune *intelligence* au niveau de la connaissance des règles du jeu d’échec, il s'agit uniquement d'un visualiseur d'un état donné.

  Ce code est déjà complet, il ne fait pas partie du cadre du projet à compléter. Il peut être utilisé en tant qu'exécutable simple.
  Pour les étudiants intéressés, le code source vous est entièrement fourni. L'ensemble du code est écrit en C, et il utilise la librairie graphique OpenGL ainsi que le gestionnaire de fenêtre Glut (pour la version 3D).

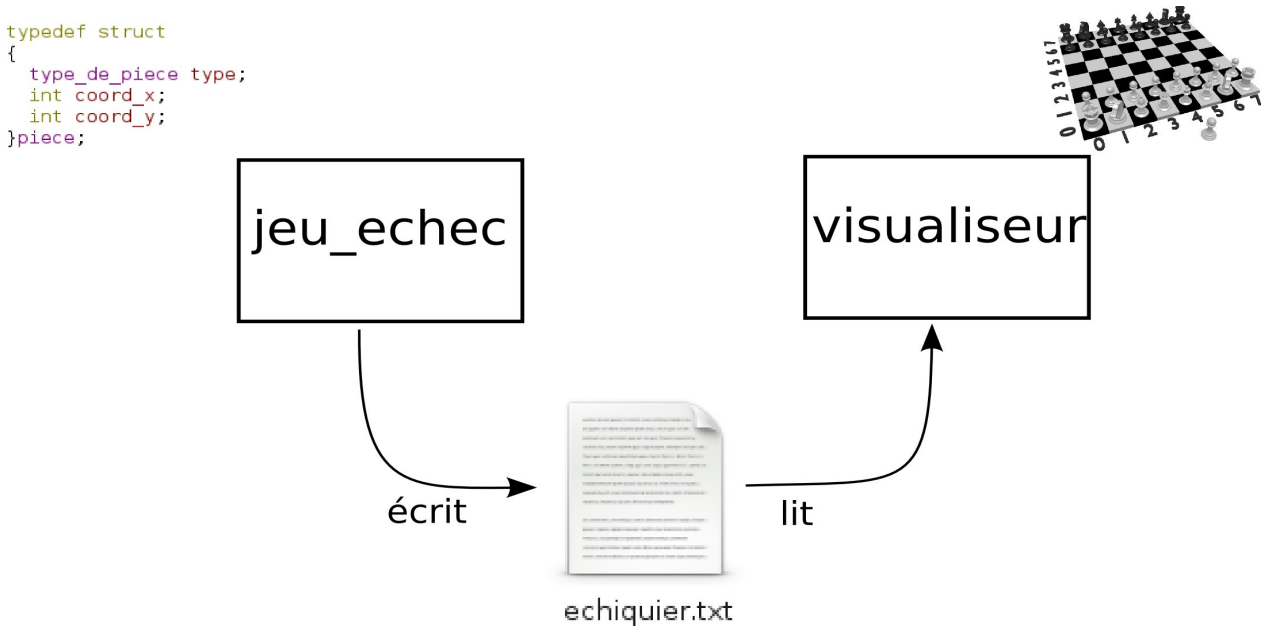
- Le répertoire **jeu_echec**:
  Contient l'ensemble des fichiers permettant de manipuler effectivement le jeu d’échec.

  Il contient le code permettant de
  - stocker l'état d'un jeu d’échec,
  - manipuler le jeu,
  - analyser le suivi des règles du jeu lors de la demande d'une pièce de l'échiquier,
  - communiquer avec l'utilisateur en ligne de commande.

  C'est cette partie du code que vous allez compléter lors de ce projet.

La communication entre ces deux programmes est réalisée à l'aide d'un fichier texte écrit sur le disque dur.

- L'exécutable du jeu écrit à chaque coup dans ce fichier.
- Le visualiseur vient quant à lui lire ce même fichier lorsqu'il est modifié.



Organisation du projet:

Le code source du projet est réparti en différent fichiers.
Chaque fichier représente un niveau **d'abstraction** spécifique.

Au plus haut niveau d'abstraction, l'utilisateur peut définir:

- Le déplacement d'une pièce d'une coordonnée (x,y) vers une autre.
- L'initialisation de l'échiquier à une configuration de départ.
- La sauvegarde, ou le chargement d'un échiquier.

Ces fonctions d'appels de haut niveau qui permettent de manipuler l'échiquier sont situées dans un un fichier spécifique dénommé **API** (Application Programming Interface).

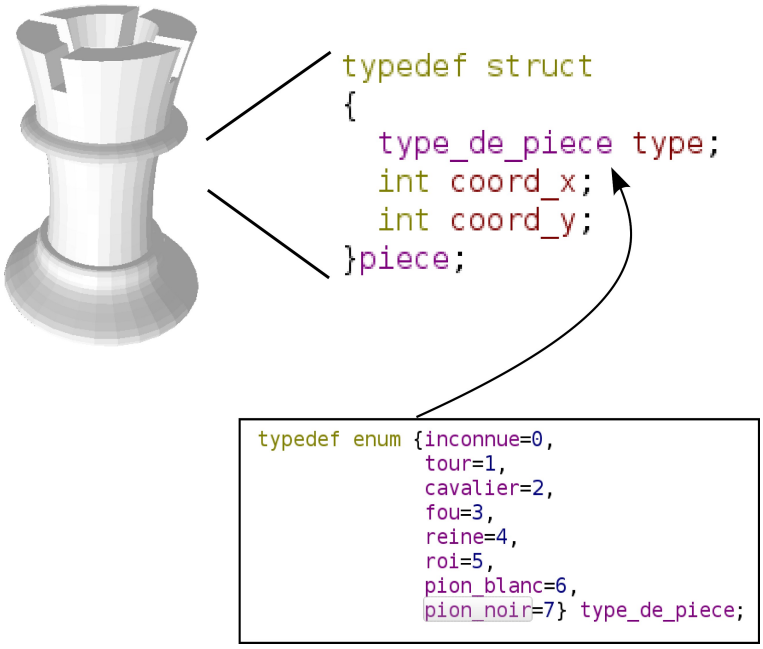
Au niveau d'abstraction plus bas, nous viendrons spécifiquement stocker une pièce en mémoire sous forme de struct, et un jeu sera formé d'un tableau de pièces.

Le détail de ces niveaux d'abstractions est fourni-ci après.

Piece

Le premier niveau d'abstraction est celui représentant une pièce du jeu.
Une pièce est une structure C (struct) contenant:

- Une coordonnée = 2 entiers, l'un pour x, l'autre pour y. Ces coordonnées sont des entiers et doivent être comprises entre 0 et 7.
- Un type définissant de quelle pièce il s'agit. Le type est donné sous forme d'énumération entre (tour, cavalier, fou, reine, roi, pion blanc/noir).



Prenez bien note que la structure unitaire d'une pièce contient ses coordonnées sur l'échiquier.
Une tour aux coordonnées (0,0) et une tour déplacée aux coordonnées (7,0) sont donc deux pièces différentes.

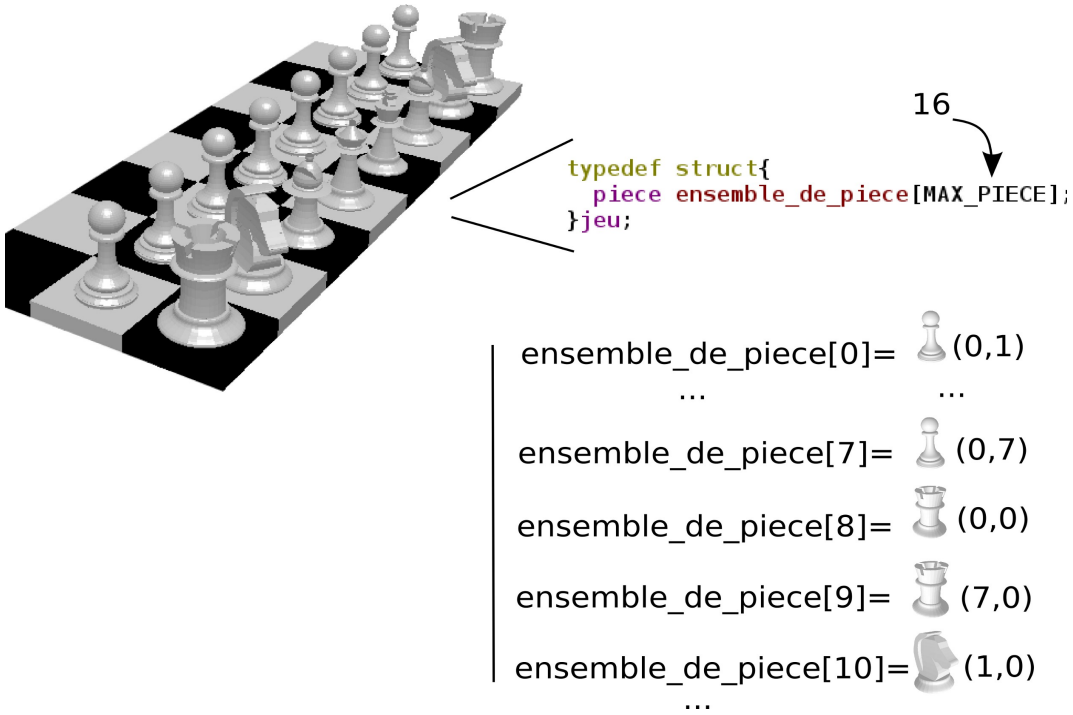
Vous compléterez les fonctions de ce niveau d'abstraction lors de ce projet.

Jeu

Le second niveau d'abstraction est un jeu de pièce.
Un jeu représente l'ensemble des pièces d'un joueur, soit les pièces blanches, soit les pièces noires.
Un jeu est donc un conteneur pour un ensemble de pièces. Dans notre cas, on choisit d'implémenter ce conteneur sous forme de tableau statique.

Un jeu contient au maximum 16 pièces. Le tableau aura donc une dimension d'au moins 16.
Pour simuler les pièces supprimées du jeu, nous placerons des pièces au contenu invalide en fin de tableau.

Il s'agit par contre d'un aspect bas niveau de l'implémentation. Les fonctionnalités de haut niveau doivent pouvoir manipuler un jeu d'échec de manière similaire, même si le stockage est réalisé de manière différente.
On veillera donc à mettre en place des fonctionnalités génériques telles que:
parcourir l'ensemble des pièces valides d'un jeu, trouver ou manipuler une pièce existant à des coordonnées particulières.
De cette manière, seul les fonctions de ce fichier viendront directement manipuler le tableau, et non les fonctions extérieures à ce niveau d'abstraction.

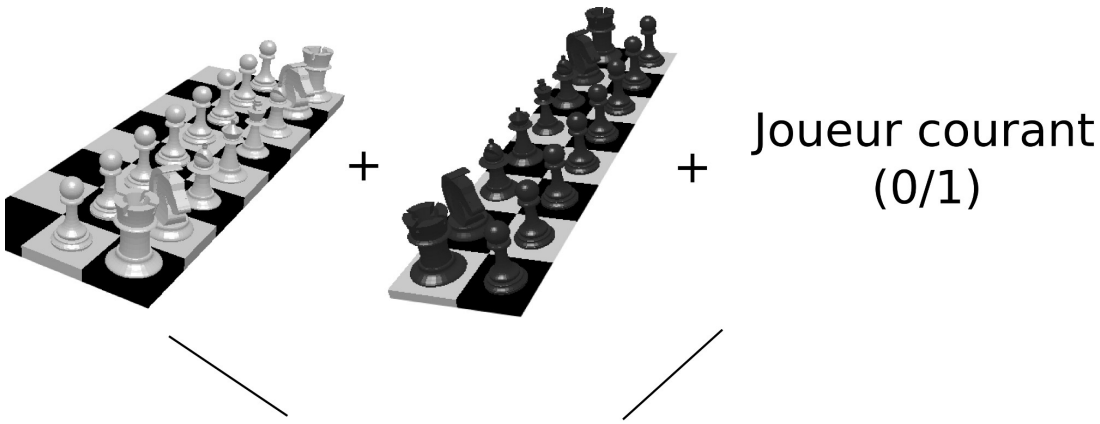


Vous compléterez les fonctions de ce niveau d'abstraction lors de ce projet dans les premières séances.

Echiquier

Un échiquier est formé de deux jeux et du joueur courant. Il s'agit donc d'une structure C qui contient:

- le jeu de pièces blanches
- le jeu de pièces noires
- l'indice du joueur courant (0 pour joueur pièces blanches, 1 pour joueur pièces noires).




```
typedef struct
{
    jeu jeu_de_piece_blanc;
    jeu jeu_de_piece_noir;
    int joueur_courant;
}echiquier;
```

Note: Attention, un échiquier n'est **pas** un tableau 2D contenant des pièces ou du vide!
Il s'agirait d'une autre structure de données possible! L'algorithmique de parcours d'un jeu donné serait différent.

Vous compléterez les fonctions de ce niveau d'abstraction lors de ce projet dans les premières séances.

Echiquier déplacement:

Echiquier déplacement est une abstraction qui permet ou non de valider le déplacement d'une pièce de l'échiquier en fonction des règles du jeu d'échec.
C'est en quelque sorte la partie *intelligence* du jeu.
Notez qu'il ne s'agit pas de la modélisation d'une entité physique telle qu'une pièce du jeu.

Contrairement aux structures de stockage précédentes, cette abstraction est supposée être de haut-niveau. Elle n'a pas à interagir directement avec le tableau statique contenant les pièces, et doit pouvoir être indépendante du type de stockage.

Elle doit offrir par contre une interface simple telle que: déplacement de la pièce située aux coordonnées (x0,y0) vers (x1,y1).
Dans le cas où le déplacement est permis par les règles du jeu, celui-ci peut être effectué.
Dans le cas où le déplacement est non permis, une structure d'erreur spécifique est mise à jour permettant au joueur de comprendre en quoi ce coup est invalide.

Cette structure d'erreur contient:

- Le type d'erreur sous forme d'énumération (ex. Absence de déplacement, pièce bloquante, …)
- Les coordonnées qui sont potentiellement à l'origine de l'erreur (tel que les coordonnées d'une pièce bloquant un déplacement).

Vous compléterez la totalité des fonctions de ce niveau d'abstraction lors de ce projet dans les dernières séances.

API echec:

L'API est l'interface de plus haut niveau fournie par une librairie. C'est cette interface qui est utilisée lorsqu'un développeur veut utiliser une librairie donnée afin de l'interfacer avec son propre code.
Dans notre cas, l'API représente le niveau d'abstraction de l'intention (ou la main) de l'utilisateur vis à vis du jeu d'échec.

Dans notre cas, l'API propose les fonctions de:

- déplacement de pièce d'une coordonnée à une autre,
- initialisation du jeu d'échec,
- sauvegarde de l'état du jeu dans un fichier,
- lecture de l'état du jeu à partir d'un fichier.

Toutes les variables donnée en tant qu'argument des fonctions de l'API doivent être génériques (ex. Coordonnées x et y).
Elle ne doivent pas dépendre du type de la structure de stockage par exemple.

Dans notre cas, l'API propose également une fonction qui lance une communication interactive avec l'utilisateur en ligne de commande.

Vous compléterez la gestion globale de déplacement d'une pièce dans l'API lors de ce projet.

Entree sortie :

Entree sortie est un niveau d'abstraction permettant de faire le lien entre l'échiquier existant en RAM et un état d'échiquier stocké sur le disque dur.
Il propose une interface de lecture et d'écriture dans des fichiers du disque.

Il s'agit principalement de concentrer la partie spécifique à l'écriture sur le disque dans une abstraction spécifique afin d'offrir une interface de sauvegarde et de chargement simple pour les autres partie du programme.

De même, il s'agit d'une structure qui ne doit pas dépendre du type de stockage du jeu d'échec en interne. On considère que les appels de cette abstraction sont de haut-niveau.

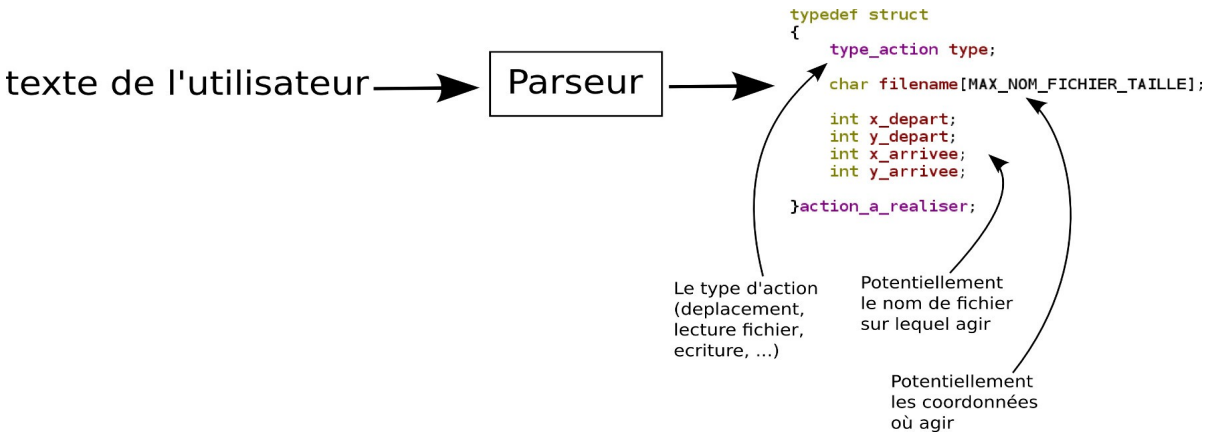
Vous écrirez le code de sauvegarde de l'état de l'échiquier dans un fichier lors de ce projet.

Parseur mode interactif:

Tout comme entree_sortie, ce niveau d'abstraction vient principalement décharger l'API de la problématique de la validation et de l'interprétation d'une chaîne de caractère donnée par l'utilisateur lors du dialogue en ligne de commande.

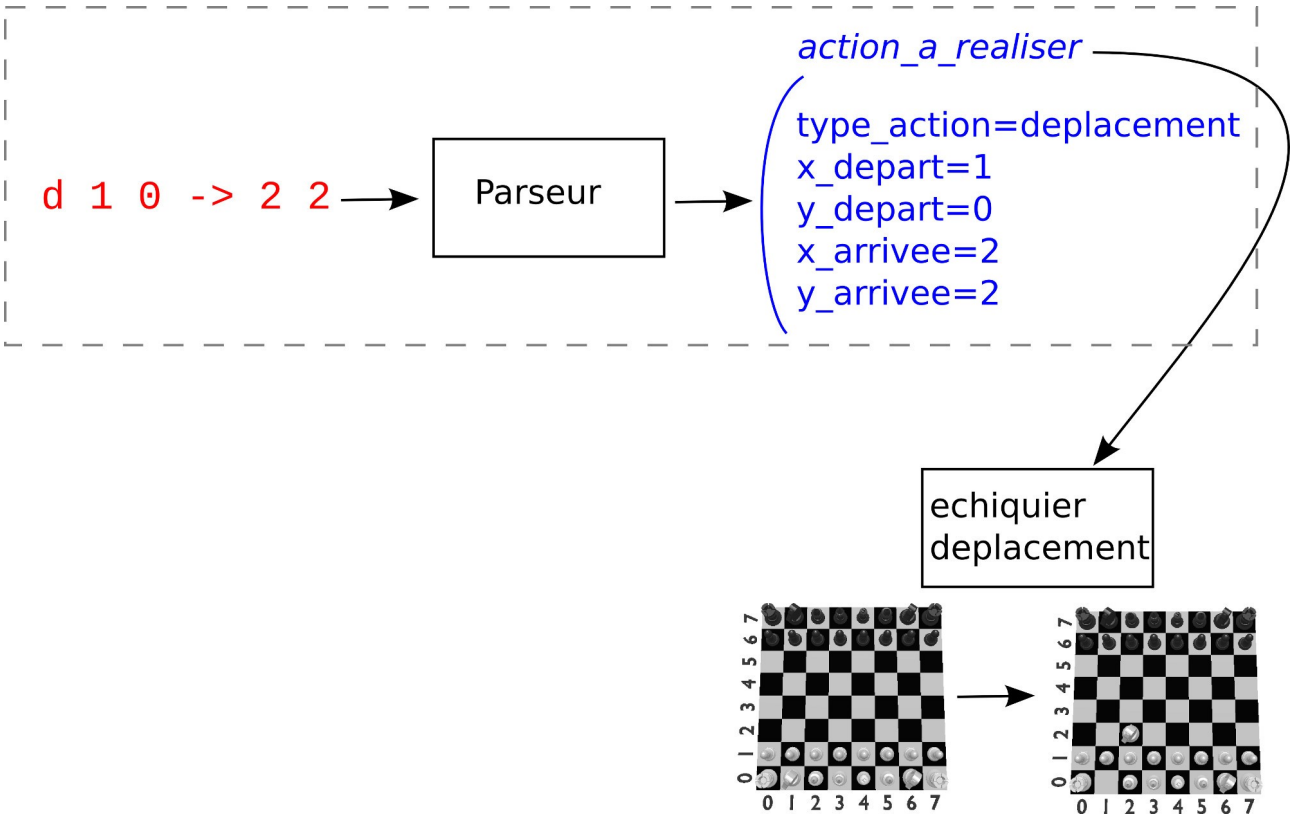
Le parseur vient tout d'abord analyser si la chaîne de caractère est valide au niveau de sa syntaxe.
Ensuite, il vient interpréter ce message.
Enfin, il renvoie une structure contenant les informations de ce message.

Note: Parseur est un terme commun d'informatique désignant l'analyse syntaxique d'un texte.
(ex. Parseur xml, parseur de ligne de commande, etc).



Exemple:

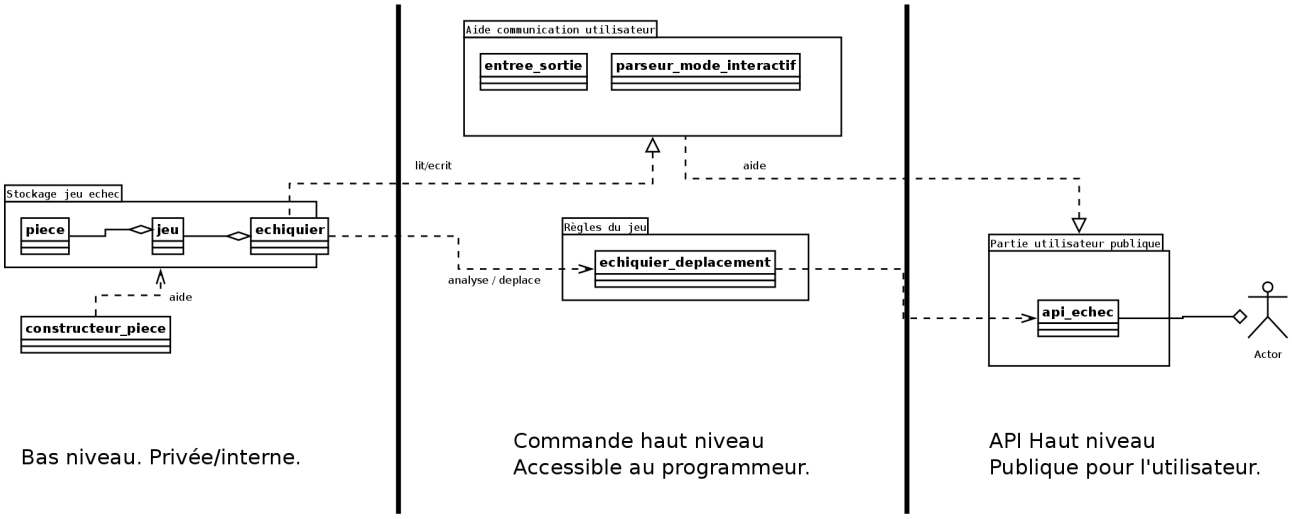
Entrée utilisateur =
`> d 1 0 -> 2 2`



Le parseur est déjà entièrement écrit lors de ce projet.

Diagramme des niveaux d'abstractions:

Le diagramme suivant résume l'organisation des différents niveaux d'abstraction du projet ainsi que leurs relations.



Plagiat et Honnêteté intellectuelle
CPE

- Tout plagiat (copie sans citation de sources) est interdite sur l'ensemble des documents rendus.

- L'honnêteté intellectuelle consiste à citer les sources directes ou indirectes d'inspiration qui ont permis de réaliser un travail.

Règle fondamentale : **Citez vos sources.**

Il est essentiel que, lors du déroulement des examens, comme lors de toutes les activités pédagogiques, les élèves fassent preuve du comportement responsable et professionnel attendu de futurs ingénieurs, dans un souci permanent d'éthique et d'honnêteté.

Citation du règlement des études de CPE :
(`https://e-campus.cpe.fr/mod/resource/view.php?id=36`)

1 Dénomination

Le plagiat consiste à utiliser tout ou partie du travail de quelqu'un d'autre sans le mentionner.

Remarque sur la quantité de la copie : La quantité de plagiat peut varier de l'ensemble d'un document à une seule ligne (code et/ou texte). Il n'y a pas de notion de *quantité admissible*.

Remarque sur les licences :
- Utiliser le contenu d'un document *libre* sous licence *libre* telle que GPL [1] dans un travail implique que votre travail passe en totalité sous licence *libre* également (ex. Utilisation de certaines images Wikipédia).
  C'est à dire : distribution intégrale du code source avec l'executable, et mention de la licence libre sur chaque fichier.
- Utiliser le contenu d'un document sous copyrights sans permission explicite de l'auteur est interdit et peut entrainer des sanctions pénales et financières.

Dans le cadre de travaux d'étudiants, il est toléré de pouvoir utiliser des documents sous licence libres en les citant, sans avoir à placer une notion de licence sur ce travail.
L'utilisation de documents sous copyright cités explicitement est généralement tolérée, mais vous engage à titre personnelle de vérifier que la copie n'est pas explicitement interdite.

Dans tout les cas, l'absence de citation de la source est interdite et pénalisée.

2 Utilisation de ressources existantes

- Il n'est pas interdit d'utiliser Wikipedia, ni de le citer.
- Il n'est pas interdit de recopier mot pour mot un texte existant à condition de le citer explicitement.
- Il n'est pas interdit d'utiliser le même code, le même paragraphe, ou les mêmes résultats que votre voisin à condition de citer explicitement la source.

3 Notation dans le cadre scolaire

- La copie sans citation est interdite et sanctionnée.
- La copie avec citation est autorisée mais ne rapporte pas de points.
- La copie avec citation ainsi qu'une analyse ou une extension supplémentaire commence à rapporter des points.

Vous pouvez donc utiliser à bon escient le travail d'autres personnes citées en le complétant et en l'analysant.

1. http ://www.gnu.org/licenses/gpl.html

4 Exemples de citations

Note préliminaire : Il vaut mieux citer trop de références que pas assez.

4.1 Version professionnelle :

Nous implémentons l'approche décrite par Cockburn [C05].

(à la fin du document) **Bibliographie :**

[C05] Alistair Cockburn. *Agile Software Development*. Addison-Wesley Professional, 2005.

4.2 Citation Wikipedia :

Comme le décrit la page Wikipedia, les tests d'intégration *ont pour but de valider le fait que toutes les parties développées indépendamment fonctionnent bien ensemble de façon cohérente.*.

Nous utilisons le code du tri par insertion donné sur la page Wikipedia (`http://fr.wikipedia.org/wiki/Tri_par_insertion`):

```
procédure TriInsertion(tableau T, entier n)
    pour i de 2 à n
        x = T[i]
        j = i
        tant que j > 1 et T[j - 1] > x
            T[j] = T[j - 1]
            j = j - 1
        fin tant que
        T[j] = x
   fin pour
fin procédure
```

4.3 Citation Wiki (2) :

Il nous a paru interessant de coder une courbe de Bézier en language Python. Pour cela, nous nous sommes inspirés du code présent sur ce Wiki :
`http://rosettacode.org/wiki/Cubic_bezier_curves#Python`
Que nous avons ensuite intégré dans notre visualiseur. Notez que nous avons généralisé ce code à des courbe de degrés 4 également (voir notre annexe).

4.4 Citation d'un site sous copyright :

Pour implémenter notre gestion mémoire par allocation dynamique, nous nous inspirons du code fourni sur le *site du zero* [SZERO]

(à la fin du document) **Bibliographie :**

[SZERO] `http://www.siteduzero.com/tutoriel-3-14061-l-allocation-dynamique.html#ss_part_3` , auteur : Mathieu Nebra.

Pour réaliser notre implémentation de liste chainée, nous avons utilisé le code trouvé sur le site *developpez.com*.
`http://nicolasj.developpez.com/articles/listedouble/`

4.5 Inclusion d'une image sous copyright :

Nous avons trouvé sur internet une image illustrant très bien la méthodologie d'extrême programming. L'image *X* est issue du site
`http://xprogramming.com`

4.6 Citation d'un collègue :

Suite aux discussions avec notre binôme voisin (Jean Moulin et Bernard Dupont), nous avons décidé de construire une fonction d'initialisation de notre tableau séparée du reste de l'algorithme.

4.7 Citation d'un collègue (2) :

Nous n'avons pas réussi à coder cette partie du TP/projet. Afin d'avancer et de pouvoir réaliser la partie suivante, nous avons tout de même demandé au binôme (Jean Moulin et Bernard Dupont) de nous fournir leur code.

Nous avons tout de même noté que cette fonction stockait en dur le nombre de cas à vérifier. Nous avons tenté d'améliorer ce code en introduisant un nombre de cas variable. Nous avons testé l'application de la version améliorée dans le cas où l'on considère une grille de jeu plus importante.

4.8 Citation d'un collègue dans du code :

Voir figure. 1.

```
//Code de creation de tableau_2d
// Rem. Code repris du binome Jean Dumont et Francois Rousseau
//      car permet de factoriser nos appels a malloc.
// Note: Nous aurions pu utiliser l'instruction calloc pour
//      initialiser en meme temps qu'allouer.
float* cree_tableau_2d(int taille_x,int taille_y)
{
    float *tableau=NULL;
    tableau=malloc(taille_x*taille_y*sizeof(float));

    if(tableau==NULL)
    {printf("Erreur allocation tableau %dx%d\n",
            taille_x,taille_y);exit(1);}

    //initialisation valeurs a 0
    int k=0;
    for(k=0;k<taille_x*taille_y;++k)
        tableau[k]=0.0;
    return tableau;
}

//Code de liberation memoire de tableau_2d
// Rem. Code repris du binome Christelle Beauvois
//                      et Francois Aura (groupe C).
//    Leur code est tres robuste (detection des cas d'erreurs)
//      et nous a permis de continuer a avancer dans notre
//      projet sans segmentation fault.
void suprime_tableau_2d(float **tableau_a_supprimer)
{
    //verification des entrees
    assert(tableau_a_supprimer!=NULL);
    assert(*tableau_a_supprimer!=NULL);

    free(*tableau_a_supprimser);
    *tableau_a_supprimer=NULL;

    //contrat de sortie
    assert(*tableau_a_supprimer!=NULL);
}
```

FIGURE 1 – Exemple de citation de collègues dans du code.

5 Sanctions

Toute détection d'un plagiat quelconque est signalée automatiquement à la direction des études de CPE.

Notez que la sanction de plagiat aura lieu sur toute forme de copie et sur n'importe quelle quantité. Que ce soit pour une copie intégrale d'un TP/projet, ou pour la copie d'une seule ligne d'un site internet, ou d'une seule ligne du TP voisin. Même une seule ligne copiée nécessite la citation adéquate.

Le règlement des études
(`https://e-campus.cpe.fr/mod/resource/view.php?id=36`)
indique que :

Le plagiat en examen ou lors de rendu de travaux évalués peut faire l'objet d'un conseil de discipline dont les sanctions peuvent être :

- un avertissement,
- un blâme,
- une exclusion temporaire,
- une exclusion définitive.

De plus, quand il s'agit de fraude, plagiat, ou tentative de fraude, toute sanction invalide le module concerné.

Consignes de rendus:

Veuillez lire attentivement et vérifiez à chaque rendu que vous respectez les consignes de rendus.
Il est de votre responsabilité de les vérifier.

Tout rendu ne vérifiant pas les consignes ne sera pas traité.
Tout rendu non traité et noté obtiendra la note de 0.

Noms du répertoire racine:

Votre répertoire contenant le projet doit se nommer:
`nom1_nom2_projet_echec/`

avec:
`nom1`: le 1er nom de votre binôme (uniquement le nom, pas le prénom).
`nom2`: le 2eme nom de votre binôme (uniquement le nom, pas le prénom).
Respectez l'ordre alphabétique afin de rester dans un ordre cohérent au fur et à mesure des dépôts.

Si vous êtes en monôme, omettez le `nom2`.

Noms de répertoires et de fichiers:

Dans vos noms, ainsi que dans tous noms de fichiers ou de répertoires, vous n'utiliserez que les caractères ASCII de base. C'est à dire les caractères situés entre a-z. L'underscore (_) est autorisé et permet de séparer les mots.

Notez qu'en particulier, sont interdits:

- Les accents
- Les espaces
- Les cédilles
- Les majuscules.

ex. de noms de répertoires ou de fichiers non conformes:
```
Umberto_Chala/
mon projet/
Audré_humbert/
```

à remplacer par:
```
umberto_chala/
mon_projet/
audre_humbert/
```

But: Rendre vos fichiers et répertoires **portables**.
Les caractères spéciaux, majuscules, et accents peuvent posséder des encodages différents suivant les systèmes (ex. Passage Windows/Linux). Les espaces sont excessivement lourds à traiter en ligne de commande.

Dépots de fichiers:

Lors d'un upload sur le dépôt de fichier, vous déposerez de manière systématique une archive **tar.gz** du même nom que le répertoire archivé.
Dans votre cas, votre archive doit se nommer:

```
nom1_nom2_projet_echec.tar.gz
```

Une fois décompressée, cette archive doit donner le répertoire:
`nom1_nom2_projet_echec/`

Attention: Une archive .zip (ou .rar) renommée en .tar.gz n'est pas une archive tar.gz !
Ces archives ne seront pas traitées.

Rappel:

Pour archiver un repertoire <REP> en ligne de commande, on utilisera la syntaxe:
`$ tar cvfz <REP>.tar.gz <REP>`

Pour décompresser une archive NOM.tar.gz en ligne de commande, on utilisera la syntaxe:
`$ tar xvfz NOM.tar.gz`

Attention: Il est de votre responsabilité de vérifier l'intégrité de votre archive.
Une archive invalide ne sera pas traitée.
Pour vérifier votre archive, il est conseillé de décompresser celle-ci et de vérifier son contenu avant son envoi.

Organisation de vos répertoires:

Vos répertoires doivent présenter une structure claire et standard.
On organisera donc une structure en sous-repertoire tel que, au moment du rendu vous aillez :

un répertoire **`src/`** (comme source) contenant uniquement les fichiers sources de votre projet.
un répertoire **`bin/`** (comme binaire) contenant uniquement le (/ou les) fichier(s) binaire(s) executable(s) de votre projet
un répertoire **`test/`** contenant uniquement les fichiers de tests du projet (voir séance sur les tests)
(optionnel : un répertoire **`lib/`** (comme librairie) contenant uniquement les librairies de votre projet. (.so, .a))

Voici un exemple de schéma valide en début de projet:

```
nom1_nom2_projet_echec/src  /piece.h
                             piece.c
                             jeu.h
                             jeu.c
                             ...
                      /bin  /jeu_echec
                      /lib  /libecriture_fichier.a
                      /test /
```

Le répertoire **`src/`** contiendra uniquement les fichiers textes du code source. C'est à dire, les .c, les .h, potentiellement le Makefile et/ou les scripts de compilation.
Il ne doit pas contenir de fichiers temporaires, ou de binaires.
En particulier:

- Pas d'exécutables
- Pas de fichiers objets (.o)
- Pas de fichiers cachés (commençant par . , ou terminant par ~) => faites View/Afficher_les_fichiers_cachés dans votre exploreur.

Lorsqu'il vous est demandé de déposer un fichier exécutable, vous placerez celui-ci dans un répertoire spécifique **`bin/`**
Si vous avez des librairies binaires (.so ou .a), elles seront placés dans le répertoire `lib/`
Si vous avez des scripts de tests, ils seront placés dans le répertoire `test/`

Fichiers:

Tous vos fichiers (.c, .h, Makefile, tests, …) doivent impérativement contenir vos noms, prénoms, et groupe suivant la syntaxe suivante:

```
//#Nom1: ZZZ
//#Prenom1: ZZZZ
//#Nom2: ZZZ
//#Prenom2: ZZZZ
//#Groupe: Z
```

(dans les tests et Makefile, les // seront omis et la ligne débutera par le caractère #).
Dans ces lignes, vous remplacerez les caractères Z, par les chiffres et numéros adéquates. Vous prendrez soin de respecter la syntaxe débutant par les symboles spécifiques //#.

N'oubliez pas que les accents, cédilles et espaces ne sont pas autorisés.

Compilation et execution:

Tous programme rendu doit compiler.
Tous programme ne compilant pas entraine la note de 0.

Si vous n'arrivez pas à corriger une erreur de compilation, **commentez** la partie provoquant l'erreur et ajoutez des commentaires de manière adéquate.

Vos programme ne doivent pas avoir d'erreurs mémoires.
Vous disposez des outils de debugs (gdb, Valgrind, …) permettant de détecter les fuites mémoires.
Si un programme contient une erreur mémoire (souvent finissant par une erreur de segmentation, ou Seg Fault) que vous n'arrivez pas à régler, commentez la partie générant cette erreur et ajoutez des commentaires de manière adéquate.

Fraude et copie:

Toute fraude est interdite.
Toute copie sur un binôme voisin sans le citer, ou à partir de sources externes (livre, internet, ...) sans citation entraîne automatiquement l'invalidation de votre module et l'alerte de la direction des études.

Il est indispensable de citer explicitement dans votre code/compte-rendus/scripts toute copie/inspiration d'un binôme voisin ou de toute autre source externe.

Notez qu'il n'y a pas de taille minimale admissible de copie. Même la copie de 3 lignes de codes impose de citer vos sources.

Conclusion:
- Copier **en citant** est autorisé.
- Copier sans citer n'est pas autorisé.

Sauvegarde de vos donnés:

Vous êtes **responsables** de sauvegarder votre projet au fur et à mesure sur différent supports.
Les supports aisément accessibles dans le monde informatique sont, par exemple:

- Des dépôts sur internet (site perso, dropbox, …)
- Des clés USB
- Des CD/DVD
- Des disques durs externes
- Gestion de version (github, sourceforge, ...)

Notez que sur les PC de CPE vous n'avez pas accès au lecteur CD pour des questions de sécurités.

Attention: La perte du projet complet pour cause de crash d'un ordinateur personnel ne sera pas considéré comme une excuse autorisée.
Faites des sauvegardes régulières sur d'autres supports. Tous disque dur d'ordinateur est un support temporaire et est voué être inutilisable au bout de quelques mois/années.

Respect des délais:

Les dates de rendus sont indiquées sur les dépôts du e-campus.
Rendez à temps vos travaux.

Dans le cas d'une panne de serveur ou d'une incertitude, envoyez vos rendus par mails aux enseignants concernés (+incluez dans tous les cas en copie damien.rohmer@cpe.fr).

Rappels des mails:
damien.rohmer@cpe.fr
martine.breda@cpe.fr
nathael.pajani@cpe.fr
mohamed.sallami@cpe.fr

En cas de rendu en retard, une pénalité de 1 point par jour de retard sera mise en place.

3ETI, Developpement logiciel en C

Règles de codage.

Règles générales d'organisation et de lisibilité:

Chaque niveau **d'abstraction** est contenu dans un fichier distinct.
Les **prototypes** (ou signature) des fonctions sont écrites dans un fichier **.h**
L'implémentation de ces fonctions sont écrites dans un fichier **.c** du même nom.

Afin de privilégier la **lisibilité**, les variables seront déclarées le plus **localement** possible. De préférence juste avant leur zone d'utilisation (et non toutes au début de fonction, *norme C99*).

Initialisation des variables:

Afin de minimiser l'impact des erreurs, toutes les variables seront initialisées dès leur déclaration.
- Soit à leur valeurs finale si celle-ci est connue.
- Soit à une valeur par défaut (0 pour un entier, NULL pour un pointeur).

Cas particulier important:
A tout endroit de votre programme, tout pointeur doit soit pointer vers une zone valide, soit pointer vers NULL.

ex1. INCORRECT
```
int *g;
ma_fonction(g);
```

ex2. CORRECT
```
int *g=NULL;
ma_fonction(g);
```

Commentaires:

Chaque fichier d'en-tête contient un commentaire introductif décrivant le **rôle de l'abstraction** décrite et son utilité dans le jeu d'échec.

Chaque fonction est décrite en détail dans le fichier .h, au niveau de sa signature (ce fichier faisant partie de la librairie visible par l'utilisateur finale).
Ce commentaire doit suivre s'inscrire dans le cadre d'une **programmation par contrat** (ou assertion).

Il doit en l'occurrence décrire:
- Le **rôle** de cette fonction.
- Les **suppositions** sur les paramètres d'entrées (ces suppositions seront traduites par des assertions). Il conviendra au programmeur de les respecter.
- Les **garanties** obtenues après exécution de la fonction si les assertions sont vérifiées.

Ex. Commentaire sur la déclaration de la fonction écrivant les coordonnées (x,y) dans la structure pièce:

```
/**
 * Fonction piece_ecrit_coordonnee:
 * ********************************
 *   Ecrit les coordonnees (x,y) dans la structure de la piece_du_jeu
 *
 *   Necessite:
 *     - Un pointeur vers une struct de type piece non NULL
 *     - Deux coordonnes x et y entieres comprises sur [0,7]
 *   Garantie:
 *     - La piece est mise a jour avec les coordonnees donnees en parametres.
 */
```

Dans **l'implémentation** de la fonction (fichier .c), chaque fonction contiendra des commentaires correspondant à une méthode de **programmation par algorithme**.

Pour cela, on retrouvera dans le corps de la fonction:

- L'algorithme global suivi.
- Potentiellement, des commentaires supplémentaires devant chaque bloc non trivial.

Passage par paramètres:

Tous les types “**built-in**” (int, float, …) non modifiés par la fonction seront passés en paramètres par **copies**.

Toutes les **structures** spécifiées par l'utilisateur (struct) seront passées par **pointeurs**.

- Si la structure n'est pas modifiée par la fonction, elle sera qualifiée par le mot clé ***const***.
- Si la structure est modifiée par la fonction, elle ne sera pas qualifiée par const.

Nommage de variables/fonctions:

Il est impératif d'utiliser des noms de fonctions **préciss et signifiants**.

Afin de bien différencier les **niveaux d'abstraction** dans l'appel des fonctions, la dénomination du fichier (ou de la struct sur laquelle la fonction agit) contient le nom de celle-ci.

Ex. Pour une fonction qui spécifie les coordonnées (x,y) d'une pièce, on aura la dénomination suivante:

```
piece_ecrit_coordonnee();
```

Lorsqu'une fonction agit spécifiquement sur une struct passée en paramètre, on passera cette structure en tant que **premier argument**.

Ex. Fonction ecrivant les coordonnees (x,y) dans la structure piece:
void piece_ecrit_coordonnee(piece *piece_du_jeu,int coordonnee_x,int coordonnee_y);

Les fonctionnalités de haut niveau qu'offre ce code à l'utilisateur final sont contenu dans **l'API** (Interface de Programmation, ou Application Programming Interface).
Les fonctions de l'API seront décrites dans les fichiers **api_echec**, et leur dénomination commencera par le même terme.

Les noms de fonctions utilisent une action ou un verbe pour décrire leur rôle.

Ex. fonction d'invalidation d'une pièce du jeu :
```
void piece_invalide(piece* piece_du_jeu);
```

Lorsqu'une fonction répond à une question par une valeur booléenne (vrai=1, faux=0), le mot clé "**est**" doit être présent dans la fonction et suivi de la question posée.

Ex. Vérification qu'une pièce du jeu est valide:
```
int piece_est_valide(const piece* piece_du_jeu);
```

Seance 1:

Découverte et mise en place des fichiers.

(durée max: 1h)

- ➔ **Observez** la structure globale du projet.
- ➔ **Retrouvez** les différents niveaux d'abstractions implémentés dans ce projet.

- ➔ **Remplissez** les cases: *Nom*, *Prenom* et *Groupes* de tous les fichiers suivant les règles indiquées dans les consignes.

Ces fichiers deviennent donc vos propres fichiers à titre nominatif. Tout fichier créé devra obligatoirement contenir cette syntaxe.

Notez qu'aucun Makefile n'est fourni. Ce sera à vous de le coder dans les séances futures.
Pour compiler le projet, on se servira du script compile.sh que l'on appellera de cette manière:

```
$ ./compile.sh
```

Note: Si le fichier compile.sh n'est pas reconnu en tant que programme exécutable, lancez la commande suivante:
$ chmod +x compile.sh
qui permet d'indiquer qu'un fichier est exécutable.

Note2 : Le symbole $ *indique une commande à lancer en ligne de commande. Ce n'est pas un caractère à taper.*

Vous disposez pour vous aider à la compréhension du projet, d'un binaire exécutable réalisant la solution minimale de ce projet. Ce binaire fonctionnera jusqu'à la troisième séance, au delà il cessera de fonctionner.

Ecriture de votre première fonction pas à pas, et méthode de développement par contrats.

(durée max: 3h)

Manipulation de piece:

Le fichier *piece.h* se compose de deux parties:
1/ La *première partie* comporte la **déclaration des types** et des **enumerations** associées à l'abstraction de piece.
2/ La *seconde partie* comporte les **signatures** (en-tête) des fonctions associées à piece.

- ➔ Observez la structure piece.
- ➔ Quelles valeurs peut prendre un type_de_piece ?

Vous devez être capable d'utiliser une enumeration, demandez à un enseignant si vous ne comprenez pas cette partie.

- ➔ Dans la fonction main, tapez:

```
int main()
{
   piece p;
   p.type=tour;
   p.coord_x=5;
   p.coord_y=8;
   printf("%d %d %d\n",p.type,p.coord_x,p.coord_y);

   return 0;
}
```

- ➔ Expliquez l'affichage obtenu.

Vous devez être capable d'afficher des variables de type:
int (%d) ; float/double (%f) ; chaine de caractere (%s)
à l'aide de ***printf***. Demandez à un enseignant si vous ne comprenez pas cette partie.

- ➔ Quelles valeurs peuvent être affichées suivant le type de piece?
- ➔ Que peut on conclure si la commande suivante affiche “2” ?

```
printf(“%d \n”,p.type); //p etant une struct de type piece.
```

La fonction "*nom_type_de_piece*", permet de réaliser la correspondance entre un type_de_piece et sa chaîne de caractère.

➔ Tapez et commentez le résultat de l'appel suivant:

```
int main()
{
    piece p;
    p.type=reine;
    printf("Ma piece est une %s \n",nom_type_de_piece(p.type) );

    return 0;
}
```

Implémentation naive:

La fonction *piece_ecrit_coordonnee* permet d'ecrire les coordonnees (x,y) dans la structure de *piece* passée en paramètre.

Notez:
- Que le mot **piece** est répété en début de fonction afin de renseigner sur quelle **structure** agit cette fonction.
- Que le **premier paramètre** de cette fonction est un **pointeur** vers une **struct piece** (voir règles de codage).
- Que les paramètres suivants (les coordonnées à compléter) sont passées par **copie**.
- Que les commentaires avant la signature de la fonction décrivent la fonction suivant une règle de type "**programmation par contrat**".

➔ Implémentez la fonction de cette manière:

```
void piece_ecrit_coordonnee(piece *piece_du_jeu,int
nouvelle_coordonnee_x,int nouvelle_coordonnee_y)
{
    piece_du_jeu->coord_x=nouvelle_coordonnee_x;
    piece_du_jeu->coord_y=nouvelle_coordonnee_y;
}
```

➔ Quels sont les défauts de cette implémentation?

➔ Ecrivez les lignes suivantes dans le fichier main:

```
int main()
{
```

```
    piece p;
    piece_ecrit_coordonnee(&p,5,6);
    printf("%d %d\n",p.coord_x,p.coord_y);

    piece_ecrit_coordonnee(&p,5,12);
    printf("%d %d\n",p.coord_x,p.coord_y);

    piece *p2;
    piece_ecrit_coordonnee(p2,3,4);
    printf("%d %d\n",p2->coord_x,p2->coord_y);

    return 0;
}
```

- ➔ Qu'obtenez vous?
- ➔ Qu'est ce qui n'a pas été respecté vis à vis du contrat donné en commentaires dans le fichier .h ?

Apprendre à debugger:

- ➔ Initialisez désormais p2 au pointeur NULL.
  `piece *p2=NULL;`
  Est-ce que cela permet de debugger l'erreur ?

- ➔ En ligne de commande, lancez désormais le programme suivant:
  `$ gdb ./jeu_echec`
  Puis tappez:
  `(gdb) run`

- ➔ Observez les dernières lignes. Cela indique-il l'endroit de l'erreur?
  Pour obtenir d'avantage de précisions, tapez “`where`”.

Vous devez être capable de debugger vos programme vous même. Lorsque vous demandez de l'aide, vous devez au moins avoir localisé votre erreur mémoire. Demandez à un enseignant si cette partie n'est pas claire.

Cette méthodologie permet de déterminer où ont lieu les erreurs mémoire. Vous devez toujours être capable de localiser où a lieu une erreur mémoire.
Notez qu'il est indispensable de compiler avec le mode “debug” (option -g de gcc) pour obtenir ces informations.

Autre méthode de debug:

➔ Tappez
```
$ valgrind ./jeu_echec
```

➔ Observez la ligne "*Invalid write*": celle-ci indique le type et le lieu de l'erreur mémoire.

➔ Commentez l'appel à:
```
// piece_ecrit_coordonnee(p2,3,4);
// printf("%d %d\n",p2->coord_x,p2->coord_y);
```

➔ Recompilez et relancez votre programme.
Relancez:
```
$ valgrind ./jeu_echec
```

➔ Vérifiez que cette fois, aucune fuite mémoire n'est detectée.

Valgrind est un programme permettant (entre-autres) de détecter des fuites mémoires dans l'exécution d'un programme. Il détecte également les fuites mémoires n'aboutissant pas forcément à une "seg-fault".

Pour aller plus loin:
Il permet également de vérifier que toute mémoire allouée dynamiquement est bien dé-allouée avant de quitter le programme.

Code évitant les bugs:

Reprenez l'implémentation de la fonction piece_ecrit_coordonnee.
L'erreur mémoire peut être évitée lorsque deux conditions sont remplies:
- Le pointeur invalide est initialisé à **NULL**.
- La fonction vérifie que le pointeur qu'elle utilise n'est pas **NULL** (et donc valide).

Il est donc possible de modifier votre fonction pour prendre en charge cette fonctionnalitée:

➔ Complétez votre fonction avec cette syntaxe:

```
void piece_ecrit_coordonnee(piece *piece_du_jeu,int nouvelle_coordonnee_x,int
nouvelle_coordonnee_y)
{
    if(piece_du_jeu==NULL)
    {
       printf("Erreur pointeur NULL\n");
       exit(1);
```

```
    }

    piece_du_jeu->coord_x=nouvelle_coordonnee_x;
    piece_du_jeu->coord_y=nouvelle_coordonnee_y;
}
```

Note: *exit()* permet de quitter votre programme en renvoyant ici le code d'erreur 1 à la ligne de commande
Pour plus d'informations, tapez:
`$ man 3 exit`
en ligne de commande.

➔ Relancez votre code en ré-appelant l'appel sur **p2** initialisé à NULL.
Cette fois votre programme détecte bien l'erreur.

Analyse:

Il existe 2 désavantages au code que nous venons de mettre en place:

1- L'erreur est bien pris en charge, mais **l'endroit** précis de celle-ci n'est pas indiqué automatiquement.
Dans un projet pouvant contenir des milliers de lignes de codes et des centaines de fonctions, il est nécessaire de connaître plus de précisions pour debugger rapidement.

Pour aller plus loin:
Des macros spécifiques au C permettent de fournir des sorties automatiques sans avoir à ecrire manuellement le nom des fonctions:
Remplacer le printf affichant l'erreur par le suivant et observez la sortie:
printf("Erreur NULL pointeur: fonction %s \nLigne %d du fichier %s\n",__FUNCTION__,__LINE__,__FILE__);

2- La condition de validité "if" est exécutée en **permanence**, gaspillant de la ressource machine pour une condition normalement respectée.
Cela peut pénaliser le temps d'exécution de fonctions appelées un très grand nombre de fois. (ex. Récupération d'une valeur dans un tableau avec vérification des bornes plusieurs millions de fois).

Pour pallier à ces désavantages, il existe la fonction "**`assert`**" dont le rôle est de vérifier les **assertions.** C'est à dire des conditions de programmations qui sont supposées vraies lors du développement du logiciel.

Les assertions:

➔ Observez la page man de **`assert`** avec la ligne de commande:
`$ man assert`

Vous devez être capable de lire une aide des pages **man** (manual). Toutes les fonctions standards du C sont dans les pages man. Demandez à un enseignant si vous ne comprenez pas la page man.

Dans notre cas, la fonction devient alors:

```
void piece_ecrit_coordonnee(piece *piece_du_jeu,int nouvelle_coordonnee_x,int
nouvelle_coordonnee_y)
{
    assert(piece_du_jeu!=NULL);

    piece_du_jeu->coord_x=nouvelle_coordonnee_x;
    piece_du_jeu->coord_y=nouvelle_coordonnee_y;
}
```

➔ Modifiez votre code en suivant le modèle donné.

La fonction **`assert`** possède deux **avantages**:

1- Elle indique la **localisation** de l'erreur de codage rencontrée explicitement.

➔ Testez le programme et observez la ligne d'erreur. Vous devez être capable de la comprendre.

2- La vérification de la condition est réalisée uniquement lorsque le programme est compilé en mode de développement (dit de “debug”).
Lorsque la librairie/logiciel est terminé (on parle alors de mode “release”), la vérification n'est plus effectuée et évite le sur-coût de son évaluation.

➔ Pour indiquer que l'on ne souhaite plus évaluer les assertions, ajoutez l'option de compilation **-DNDEBUG** à chaque ligne de compilation (fichier compile.sh). Observez que l'erreur mémoire apparaît à nouveau.

Vous devez être capable de modifier les paramètres de compilation, appelez un enseignant si vous avez des questions.

Note: Lors de la compilation, l'ajout de l'option -D<X> est similaire à l'ajout de la ligne #define <X> en début de chaque fichier. Ici, cela revient à ajouter la ligne #define NDEBUG en début de chaque fichier. NDEBUG signifiant: No Debug.

Note importante: La fonction **`assert`** est particulièrement utile dans un but de debug lors du développement d'un code. Elle peut être largement utilisée de manière systématique afin de vérifier les certifications des fonctions.
Elle ne doit être utilisée que dans ce cadre là. C'est à dire pour détecter:

- Des **erreurs de codage** qui ne doivent jamais arriver une fois le projet fini.
- Des erreurs qui impliquent forcément un arrêt **immédiat** du programme suivit d'un debug du code.

En l'occurrence, on n'utilisera pas la fonction `assert` pour traiter:

- Des erreurs critiques qui doivent également être traitées dans le **code finalisé** compilé en mode "release".
  ex. Verification de l'espace disque ou RAM restant lors du remplissage d'une base de données, etc)
- Des erreurs non critiques devant être gérées spécifiquement pouvant avoir lieu dans la **version finale** du logiciel.
  ex. Entrée utilisateur incorrecte, chemin vers un répertoire inexistant, …

Vérification des contrats:

Le commentaire de la fonction écrit dans le fichier .h indique deux conditions sur les paramètres d'entrée. La seconde condition n'est cependant pas vérifiée dans l'implémentation de la fonction.

En particulier, la ligne:
```
piece_ecrit_coordonnee(&p,5,12);
```
Fonctionne correctement, alors qu'elle ne **vérifie pas le contrat** donné en commentaires.

Note importante sur le développement logiciel:
Les erreurs dites "*silencieuses*" sont les plus **difficile** à détecter et debugger.
En effet, une erreur critique ou de programmation (ex. Dépassement mémoire) est automatiquement détectée par le système (Segmentation fault) ou par le compilateur.
Une erreur non détectée par le système peut se poursuivre de manière *silencieuse* jusque dans le logiciel finalisé et aboutir à des comportements étranges ou à des failles de sécurité une fois distribué.
Une bonne programmation permet de **détecter** et corriger **le plus tôt possible** le comportement inapproprié des fonctions afin de prévenir tout comportement non attendu par la suite.

Par ordre de priorité:
1- Au moment de la compilation (le mieux).
2- Par assertions à l'exécution.
3- Au moment de tests unitaires.
4- Au moment de tests d'intégration.
5- Lors de l'utilisation en production/par le client. (le pire).

Dans le cas de notre fonction, il est possible et permis d'affecter la valeur 12 à un entier (niveau compilateur).
Cette valeur est cependant incorrecte vis à vis des *coordonnées d'un échiquier*. Il est donc important de ne pas permettre la manipulation de telles coordonnées dans les fonctions de notre jeu.

Le fait que les coordonnées passées en paramètres soient comprises dans l'intervalle [0,7] font parties du **contrat** passé avec le programmeur. La certification de cette fonctionnalité peut être vérifiée par la fonction **`assert`**.

Les deux syntaxes suivantes répondent au problème:

```
assert(coordonnees_x>=0);
assert(coordonnees_y>=0);
assert(coordonnees_x<=7);
assert(coordonnees_y<=7);
```

Ou bien:

```
assert(coordonnees_x>=0 && coordonnees_y>=0 && coordonnees_x<=7 &&
coordonnees_y<=7);
```

La seconde solution possède l'avantage de se réduire à une ligne, mais l'identification d'un problème éventuel est moins précise.

Pour aller plus loin:
En C, il aurait été possible de certifier que les valeurs sont positives par le compilateur, en déclarant les coordonnées comme des "unsigned int", et non des "int". La vérification ayant uniquement lieu vis à vis du nombre 7.

Fonction de certification de coordonnées:

Il est à prévoir que cette vérification ait lieu un grand nombre de fois dans diverses fonctions. Afin d'éviter la **duplication** de ces lignes de codes dans plusieurs fonctions, nous allons centraliser l'appel à ces vérifications en créant une nouvelle fonction d'aide permettant de valider si une coordonnée est valide.

La signature de cette nouvelle fonction est la suivante:

```
/**
 * Fonction est_coordonnee_valide:
 * ************************
 *    Verifie que deux coordonnees (x,y) sont valides sur un echiquier.
 *    C'est a dire que x et y sont deux entiers compris dans l'intervalle [0,7].
 *
 *    Necessite:
 *      - Deux coordonnees de type entieres
 *    Garantie:
 *      - Un retour valant 1 si les coordonnees sont valides.
 *      - Un retour valant 0 si les coordonnees ne sont pas valides.
 */
int est_coordonnee_valide(int coordonnee_x,int coordonnee_y);
```

Note: La condition nécessaire du type entier est directement vérifiée par le compilateur. Il n'est donc pas nécessaire d'ajouter de ligne de code spécifique.

➔ Implémentez cette fonction. Notez qu'il s'agit de la réponse à la question "*est ce que les coordonnees sont valides*". Le mot clé "***est***" est donc présent dans le nom de la fonction.

Test du bon fonctionnement de la fonction:

➔ Vérifiez la sortie de cette fonction en entrant différentes coordonnées (valides et non valides) dans la fonction main.

Exemple:

```
int main()
{
    int valeur_01=est_coordonnee_valide(0,0);
    int valeur_02=est_coordonnee_valide(1,3);
    int valeur_03=est_coordonnee_valide(7,7);
    int valeur_04=est_coordonnee_valide(-5,5);
    int valeur_05=est_coordonnee_valide(5,-5);
    int valeur_06=est_coordonnee_valide(-5,-5);
    int valeur_07=est_coordonnee_valide(8,5);
    int valeur_08=est_coordonnee_valide(8,12);
    int valeur_09=est_coordonnee_valide(14,12);
    int valeur_10=est_coordonnee_valide(14,-12);
    int valeur_11=est_coordonnee_valide(-14,12);

    printf("01: %d \n",valeur_01);
    printf("02: %d \n",valeur_02);
    printf("03: %d \n",valeur_03);
    printf("04: %d \n",valeur_04);
    printf("05: %d \n",valeur_05);
    printf("06: %d \n",valeur_06);
    printf("07: %d \n",valeur_07);
    printf("08: %d \n",valeur_08);
    printf("09: %d \n",valeur_09);
    printf("10: %d \n",valeur_10);
    printf("11: %d \n",valeur_11);

    return 0;
}
```

Pour aller plus loin:
Notez que l'on teste les 8 combinaisons possibles des valeurs fausses.
A l'aide d'une double boucle, il serait également possible de tester de manière exhaustive tous les cas valides.

➔ Utilisez désormais cette fonction pour simplifier l'appel à **`assert`** dans la fonction `piece_ecrit_coordonnee`

Le code final devra alors ressembler à:

```
void piece_ecrit_coordonnee(piece *piece_du_jeu,int nouvelle_coordonnee_x,int
nouvelle_coordonnee_y)
{
    //vérification des donnees d'entrees
    assert(piece_du_jeu!=NULL);
    assert(est_coordonnee_valide(nouvelle_coordonnee_x,
nouvelle_coordonnee_y)==1);

    // La piece du jeu recoit les nouvelles coordonnees en x et y
    piece_du_jeu->coord_x=nouvelle_coordonnee_x;
    piece_du_jeu->coord_y=nouvelle_coordonnee_y;
}
```

Notez l'ajout des **commentaires** aidant à l'explication du code.

Démarche de codage:

Notez que l'écriture d'une fonction de 2 lignes a nécessité une réflexion importante sur le **type d'erreur à traiter** et les **conditions d'utilisation** de cette fonction.

Pour chaque fonction que vous allez coder par la suite, il vous est demandé de respecter cette démarche:

- Commentaires
- Codage par contrat
- Vérification des assertions
- Tests des fonctionnalités

Votre évaluation portera en grande partie sur vos analyses, vos tests, vos commentaires et description du comportement, et respect des contrats.
Posez des questions à un enseignant si cette partie n'est pas claire.

Travail en autonomie.

(durée estimée: 2h)

- Révision cours, structure du projet, programmation par contrats, assertions. (1h)
- Avancement du code (1h)

Une pièce est **invalide** si son *type* ou ses *coordonnees* ne sont pas reconnue comme étant valide.
Pour différencier clairement une pièce invalide d'une pièce valide, la fonction `piece_invalider` place l'ensemble des coordonnées à la valeur -1 et le type à 0.

➔ **Complétez** la fonction `piece_est_valide()`.
Fonction qui renvoie 1 si la pièce est valide, et 0 sinon.

➔ **Observez** la structure jeu.
Un jeu est une structure contenant un **tableau** de pièces.
Ce tableau fait une taille fixe de 16. Ce tableau contient donc forcément toujours 16 pièces.

Pour différencier les pièces valides des pièces qui ont étés supprimés au cours du jeu, nous considérons qu'une pièce supprimée est une **pièce invalide placée en bout de tableau**.
Le début du tableau doit donc toujours commencer par un ensemble de pièces valides.

➔ **Complétez** la fonction

```
int jeu_compter_piece(const jeu *jeu_de_piece);
```

dont l'algorithme est le suivant:

```
//initialiser compteur a 0
//Pour toutes les pieces du jeu
//  Si la piece courante est valide
//     Incremente compteur
//Retourne compteur
```

➔ **Complétez** la fonction `jeu_existe_piece`.
Réfléchissez à l'algorithme, écrivez **le en commentaire** de votre fonction et **implémentez** celle-ci.

➔ **Vérifiez** vos fonction dans le main sur différents exemples.
Pour cela vous initialiserez un jeu, et vérifierez les sorties de jeu_existe_piece dans le cas où les coordonnées pointent: vers une piece, vers une case vide, en dehors des limites du jeu.

➔ **Déposez** votre projet (**avec vos tests**) sur les dépôts de fichiers avant la date limite.

Seance 2:

Complétion du code de jeu.

(durée max: 2h)

Mot clé const et pointeurs:

En respectant la méthode de **programmation par contrat**, implémentez les autres fonctions de jeu.

➔ **Implémentez** `jeu_recupere_piece`

L'agorithme de cette fonction est proche de jeu_existe_piece.

Cependant elle retourne cette fois un **pointeur** vers la pièce courante.

Le pointeur retourné est un pointeur **non constant**. C'est à dire que l'objet pointé peut être modifié une fois retourné. Cela permet par exemple d'invalider une pièce ou de modifier ses coordonnées.

➔ **Implémentez** `jeu_recupere_piece_information`

Notez que cette fonction est quasi-similaire à jeu_recupere_piece.

L'unique différence consiste à passer en argument un **pointeur constant** vers un jeu, et l'on récupère un **pointeur constant** vers une pièce.

Le pointeur constant passé en argument permet de **certifier** que cette fonction **ne va pas modifier** l'état du jeu de pièces. De même, le retour de pointeur constant permet d'assurer que la valeur de la pièce restera **inchangée**.

Ce type de pointeur permet d'accéder aux valeurs (coordonnées, type), tout en certifiant que celles-ci ne seront pas modifiées.

Le mot clé **`const`** permet d'aider à la lisibilité du code en séparant les informations qui ne seront que lues, des informations qui vont être potentiellement modifiées.

➔ **Testez** les lignes de tests suivantes dans le main.

```
#include "jeu.h"
#include "constructeur_piece.h"

void initialisation_jeu(jeu *mon_jeu)
{
    piece *pointeur_piece_courante=&(mon_jeu->ensemble_de_piece[0]);
    constructeur_piece_tour(pointeur_piece_courante,1,5);
}

int main()
{
    jeu mon_jeu;
    initialisation_jeu(&mon_jeu);

    piece *piece_pointee_1=jeu_recupere_piece(&mon_jeu,1,3);
    piece *piece_pointee_2=jeu_recupere_piece(&mon_jeu,1,5);

    if(piece_pointee_1==NULL)
        printf("En (1,3), la piece retournee vaut NULL.\n");
    printf("La piece pointee en (%d,%d) est une %s.\n",
           piece_pointee_2->coord_x,
           piece_pointee_2->coord_y,
           nom_type_de_piece(piece_pointee_2->type));

    piece_pointee_2->type=fou;
    printf("Desormais, la piece pointee en (%d,%d) est un %s.\n",
           piece_pointee_2->coord_x,
           piece_pointee_2->coord_y,
           nom_type_de_piece(piece_pointee_2->type));

    return 0;
}
```

➔ **Commentez** le résultat obtenu étape par étape.

➔ **Modifiez** maintenant les deux lignes de déclaration de piece_pointee par la syntaxe suivante:

```
const piece *piece_pointee_1=jeu_recupere_piece_information(&mon_jeu,1,3);
const piece *piece_pointee_2=jeu_recupere_piece_information(&mon_jeu,1,5);
```

- ➔ Que se passe-il lors de la compilation, quelle ligne est indiquée en erreur ?
- ➔ **Supprimez** les deux dernières instructions, puis relancez le programme.
- ➔ Concluez sur l'utilité du mot clé **`const`**.

Vous devez être capable de manipuler le mot clé const à bon escient. Si vous ne comprenez pas cette partie, appelez un enseignant.

Algorithmique:

- ➔ **Implémentez** la fonction **`jeu_supprime_piece`**. Cette fonction sera appelée lorsqu'une pièce du jeu est supprimée par le joueur adverse.

A la fin de cette étape, vous disposez des classes de base de votre échiquier. Le reste du programme consiste à vérifier le bon comportement des déplacement des pièces.
Des fonctions du type `deplace_piece` *de (x0,y0) vers (x1,y1) devront être proposées pour simplifier la gestion globale du jeu.*

Compilation.

(durée max: 2h)

Le projet vous est fourni sans Makefile.
La compilation se réalise en appelant les lignes de compilations pour chaque fichier dans le script `compile.sh`.
Ce fichier est un ensemble de lignes qui sont lues et exécutées les unes derrière les autres.

Rappels sur la compilation:

On rappelle les étapes de la compilation:

1. **Compilation**: Transformation d'un fichier **source** (.c) en fichier **binaire objet** (.o)

Cette étape est réalisée pour chaque fichier source.
Elle peut se réaliser indépendamment d'un fichier source à un autre, on appelle cela la compilation séparée.

Pour lancer la compilation d'un fichier source vers un fichier objet, on utilise la syntaxe suivante en ligne de commande:

```
$ gcc -c fichier.c -o fichier.o
```

L'option *-c* permet d'indiquer l'arrêt du processus au niveau du fichier objet.

2. **Edition de liens**: Permet de passer d'un ensemble de fichiers **binaires objets** (.o) en un seul fichier binaire **executable**.

Pour chaque exécutable, il n'y a qu'un seul appel à l'édition de liens.
L'édition de liens prend comme entrée un ensemble de fichiers objets (.o) et fournit en sortie un exécutable.
L'édition de liens peut être vue comme l'assemblage de toutes les fonctionnalités en un seul endroit.
C'est également au moment de l'édition de liens que l'on se lie aux librairies.

```
$ gcc fichier_1.o fichier_2.o -o mon_executable
```

Remarques:

L'étape de compilation peut échouer s'il existe des erreurs de syntaxe, ou des chemins d'inclusion non valides (voir préprocesseur).

L'étape d'édition de lien peut échouer si

- un prototype de fonction est déclaré, utilisé, mais le corps de la fonction n'existe pas ou n'est pas accessible.
- une librairie n'est pas valide.
- il existe plusieurs ou aucune fonction “main”.

Le processus de compilation est lui même scindé en 3 étapes visibles:

1.a) Etape de **préprocesseur**
1.b) Création de code **assembleur**
1.c) Conversion en **binaire**

1.a) Preprocesseur:

Le préprocesseur est une première passe sur un fichier de code (.c) pour le rendre traitable par le compilateur. Il consiste en 2 points principaux:

- Suppression de tout les commentaires.
- Exécution des directives commençant par le signe #.

En particulier:
* Toute commande ou variable **Macro** définie à l'aide de `#define` voit sa valeur directement écrite dans le code.
* Les appels de type `#include` sont remplacés par l'intégralité du texte situé dans le fichier pointé.
* Les commandes de types `#ifdef, #ifndef, #else`, sont executées en tant que conditionnelles.

L'étape du préprocesseur est une partie programmable en **amont** du début de compilation du code C à proprement parler.
Le code issu du préprocesseur est visualisable. Pour cela, on utilise l'option **-E** de GCC.

Par exemple, on pourra écrire:
```
$ gcc -E mon_fichier.c > mon_fichier.i
```

➔ Utilisez cette commande sur les fichiers d'exemples fournies (répertoire `01_preprocesseur/`) et répondez aux questions décrites dans ces même fichiers.

Vous devez être capable de comprendre et d'utiliser les commandes du préprocesseur. Appelez un enseignant si vous ne comprenez pas.

Pour aller plus loin :
Pour les plus avancés, répondez aux questions décrites dans le répertoire `01b_preprocesseur_avance/`

1.b) Création du code assembleur:

Une fois l'étape de préprocesseur établie, le code issue de celui-ci est transcrit en assembleur spécifique au processeur.
La sortie du code assembleur peut être visualisée grâce à l'option de `gcc: -s.`

Par exemple,
`$ gcc -S mon_fichier.c`
Génère le fichier: `mon_fichier.s` contenant le code assembleur correspondant.

1.c) Code binaire:

Une fois le code assembleur généré, celui-ci est transcrit en binaire qui va pouvoir être lu par le processeur. Ce binaire est le **fichier objet .o**

Options de gcc:

Les différentes options de gcc sont disponibles ici:
http://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

En particulier, on y retrouvera une liste exhaustive des options: de *debug*, *d'optimisation*, du *préprocesseur*, de l'*éditeur de liens*, de *l'assembleur*.

Rappel des options principales de compilation.

-g: Active les symboles de debug. Permet l'utilisation des debuggers sur l'exécutable (gdb, valgrind, …). Rend l'exécutable plus gros.
Lorsque vous développez, **utilisez toujours cette option**.

-O1 ou **-O2** ou **-O3**: Active les optimisations du code de niveau 1, 2 ou 3. Par défaut: niveau 2.
N'utilisez ces options que lorsque votre projet est terminé et lorsque celui-ci nécessite une optimisation en terme de vitesse d'exécution.

-I <PATH>: Permet d'inclure le chemin <PATH> dans les répertoires courants du projet.
Par défaut les répertoires courants sont: /usr/include/ , /usr/local/include, ...
Lorsqu'un fichier d'en tête est situé dans un repertoire autre. (ex. /home/mon_nom/mes_entete/), il est possible d'ajouter ce chemin dans le projet:
`$ gcc -I /home/mon_nom/mes_entete/ fichier.c -o mon_executable`

-D NAME=VALUE: permet de définir une variable globale NAME de valeur VALUE. (permet de se substituer à un "#define NAME VALUE" pour chaque fichier compilé.

-fPIC: permet de rendre l'adresse des fonctions indépendant de leur location. Permet la création de librairie dynamique (.so).

Options de Warnings:
Les warnings sont des aides précieuses permettant d'éviter des erreurs subtiles. Il faut toujours en tenir compte. Activez un maximum de warning lors de votre développement. Un projet final ne doit plus contenir de warning.
Une liste exhaustive est disponible ici:
http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html#Warning-Options

-Wall -Wextra: sont les options de **Warning minimales** à **toujours** placer lors de vos compilations.

D'autres warnings sont également utiles pour vous assurer que votre code suit les règles de bonne programmation:

-**Wfloat-equal**: affiche un warning lorsque deux flottants sont comparés par un opérateur d'égalité (rappel: Ne jamais comparer deux float/double avec l'opérateur ==, ou !=)

-**Wshadow**: affiche un warning lorsqu'une variable locale en cache une autre.

-**Wswitch-default**: affiche un warning lorsque l'on oublie le cas "default" lors d'un switch.

-**Wswitch-enum**: affiche un warning lors de l'oubli d'un cas d'enum sur un switch.

-**Werror**: permet de transformer chaque warning en erreur: Permet de s'assurer que son programme ne contient plus de warnings.

-**Wwrite-strings**: affiche un warning lors de la conversion d'un const char[] vers char *.

-**Wpointer-arith**: affiche un warning lors de l'ajout/soustraction sur des pointeurs de type void*

-**Wcast-qual**: affiche un warning lors d'un cast supprimant un const.

-**Wredundant-decls**: affiche un warning lorsqu'une déclaration a lieu plusieurs fois.

-**Winit-self**: affiche un warning si une variable est initialisée avec elle-même.

- **pedantic**: affiche un warning si le strict respect du standard C n'est pas respecté.

Les options de l'éditeur de liens:

-l<NAME_LIB>: permet d'inclure la librairie: lib<NAME_LIB>.a ou lib<NAME_LIB>.so au projet.

-L <PATH>: permet d'inclure le chemin <PATH> dans le chemin de recherche standard de librairie.

➔ Utilisez les différents fichiers d'exemples pour manipuler certaines options de gcc (répertoire `02_compilation/`).

Pour aller plus loin :
Pour les plus avancés, répondez aux questions décrites dans le répertoire `02b_compilation_avance/`

Travail en autonomie

(durée estimé: 3h):

- Révision chaîne de compilation + fin des exercice d'exemples (2h).
- Rendu du travail + fin de complétion du code (1h)

Seance 3:

Développement par tests.

(durée max: 2h)

Etapes d'analyse de la ligne de commande:

Une fois le projet réalisé, l'échiquier doit être commandable à l'aide de *phrases* envoyées par l'utilisateur en **ligne de commande**.

Par exemple:
`> d 1 3 -> 3 2`
Devra permettre de déplacer le pion situé en (1,3) vers la case (3,2).

Les autres commandes sont les suivantes:
`> i :` Initialisera l'état de l'échiquier à l'état initial.
`> # ceci est un commentaire:` Toute phrase commençant par # est un commentaire non analysé.
`> lit <nom_du_fichier> :` permet de lire et de charger l'échiquier contenu dans "*nom_du_fichier*".
`> ecrit <nom_du_fichier> :` permet d'écrire l'état du fichier dans "*nom_du_fichie*r".

Chaque type de commande se traduit par une fonction de **l'API** du jeu.
Par exemple, la fonction `api_echec_deplacement_piece`, permet de demander le déplacement d'une pièce.
Ces fonctions sont considérées comme de "***haut niveau***". Leur paramètres ne doit pas faire apparaître **publiquement** des détails d'implémentation ou de contrat de code spécifiques.
Ce sont les fonctions qu'un **utilisateur externe** ne connaissant pas votre projet doit pouvoir **utiliser** facilement.

Le coeur du projet consiste à permettre le traitement correct du déplacement des pièces suite à la commande **`d`**.

Plusieurs étapes ont lieu successivement:

1. **Analyse de la ligne de commande** (étape dit de ***Parsing*** de texte) qui détecte si la ligne est valide au niveau textuelle, mais pas si le coup est valide au niveau du jeu.

ex.
```
# entrée textuelle considérée comme invalide:
> d 7-> 3 4 7
#entrée textuelle considérée comme valide:
> d 14 9 -> 5 12
```

L'analyse du texte sur une ligne de déplacement valide aboutit au stockage des coordonnées de départs (x0,y0) et de destination (x1,y1) dans des entiers.
Notez que cette étape est déjà implémentée.

2. **L'analyse des coordonnées** (x0,y0) et (x1,y1) pour savoir si le déplacement demandé est valide au niveau du jeu.

Cette étape peut se décomposer de la manière suivante:
Vérification que les coordonnées sont bien dans l'intervalle [0,7].

- Vérification qu'il **existe** une pièce aux coordonnées de départ.
- Vérification qu'un déplacement **non null** est bien défini.
- Vérification qu'aucune pièce du jeu du joueur n'est positionnée sur la case de **destination**.
- Vérification que le **déplacement** est bien valide pour cette pièce et qu'aucune pièce ne bloque son trajet.

3. Le **déplacement effectif** de la pièce.
Modification de ces coordonnées, et **suppression** potentielle d'une pièce du joueur adverse.

Méthodologie de tests:

Au fur et à mesure du développement de votre projet, il est important de réaliser des tests.

- Des **tests unitaires** après l'écriture de chaque fonctionnalité importante qui vont tester une ***fonctionnalité "unitaire"***.
- Des **tests d'intégrations** après un certain nombre d'étapes qui vont tester un ***comportement global***.

Le **développement par tests** consiste à écrire les tests des fonctionnalités à vérifier **avant** l'implémentation de la fonction elle même.
Il est en effet souvent plus facile de décrire les **appels** que l'on **souhaite** et la **sortie attendue**, plutôt que de décrire directement comment la fonction va être implémentée.

Par exemple, considérons le cas de la fonction `est_coordonnee_valide()`.
Avant l'écriture de la fonction, nous aurions pu définir un certain nombre de tests que doit passer cette fonction.

Par exemple:

```
int main()
{
    est_coordonnee_valide(0,0); //doit retourner 1
    est_coordonnee_valide(1,1); //doit retourner 1
    est_coordonnee_valide(2,6); //doit retourner 1
    est_coordonnee_valide(7,7); //doit retourner 1
    est_coordonnee_valide(-1,5); //doit retourner 0
    est_coordonnee_valide(1,-1); //doit retourner 0
    est_coordonnee_valide(1,8); //doit retourner 0
    est_coordonnee_valide(8,1); //doit retourner 0
}
```

Une fois les tests écrits, ils **guident** l'implémentation de la fonction.
Ici, l'écriture des tests guident sur le type d'arguments passés en paramètres (2 entiers).

Les tests dit **unitaires** doivent vérifier de manière méthodique chaque point particulier, autant les cas **valides** que les cas **non valides.**

Par exemple:
Tester uniquement que l'appel à `est_coordonnee_valide(1,5)` renvoie 1 ne permet **pas** de conclure que la fonctionnalité est correcte.

Par contre, le code suivant permet de tester que tous les cas d'entrées valides sont traités correctement:

```
int main()
{
    int kx=0,ky=0;
    for(kx=0;kx<=7;++kx)
    {
        for(ky=0;ky<=7;++ky)
        {
            if(est_coordonnee_valide(kx,ky)!=1)
            {printf("Test KO %d %d \n",kx,ky); exit(1);}
        }
    }
    printf("Test OK\n");
}
```

Ce code permet d'afficher **exhaustivement** toute erreur détectée sur l'intervalle [0,7]x[0,7].

Ce test permet de vérifier l'état valide, par contre il faut également vérifier que l'état **invalide** est correctement détecté.
Pour cela, on peut définir 6 cas possibles:

- `coordonnée x<0 && y OK`
- `coordonnée x>7 && y OK`
- `coordonnée y<0 && y OK`
- `coordonnée y>7 && y OK`
- `coordonnée x<0 && y<0`
- `coordonnée x<0 && y>7`
- `coordonnée x>7 && y<0`
- `coordonnée x>7 && y>7`

Il est important de tester les **cas limites** avec x/y valant strictement -1 et +7.
Par exemple, ne pas tester uniquement avec le nombre 78.

Mise en place des tests de l'application:

L'application doit permettre de déplacer les pièces de manière cohérente vis à vis d'une partie d'échec.

Par exemple, la combinaison
```
> i
> d 2 1 -> 2 2
> d 3 6 -> 3 5
> d 3 1 -> 3 2
> fin
```
Doit aboutir à un état de l'échiquier **valide** et connue (3 pions ont été déplacés d'une case).

Au contraire, la combinaison:
```
> i
> d 2 1 -> 2 5
> fin
```
Doit aboutir à la détection d'un coup **invalide** (un pion ne pouvant pas se déplacer de 4 cases vers l'avant).

Notez que ces combinaisons peuvent être écrites **dès maintenant**. Dans un code bien construit, ces tests de l'application ne dépendent pas du codage interne de la structure de données utilisée pour manipuler l'échiquier.
Ces tests doivent donc pouvoir être exécutés, peu importe le code interne.
Par exemple, les tests des binômes voisins peuvent être par la suite appliqués sur votre code.

Il vous est demandé d'établir la série de tests qui viendra **valider** que votre projet réalise bien la fonctionnalité d'un jeu d'échec.
Pour cela, vous allez écrire un ensemble de tests **unitaires** et **d'intégrations**.
Ils devront valider à la fois les coups **possibles**, et également confirmer que les coups **invalides** sont bien détectés comme tels.

Vos tests devront couvrir un **maximum de cas possibles**.
Réfléchissez à une **stratégie** adéquate.

Vous garderez ces tests tout au long de votre projet. Ils vous permettront de valider votre projet et vous aideront à ne pas oublier de cas particuliers lorsque vous coderez. Si ces tests sont insuffisants, ils ne pourront pas valider clairement votre travail.

Quelques conseils:

- Ne passez pas l'ensemble de vos tests à observer le déplacement valide d'un pion à différents endroits de l'échiquier. Mais **couvrez** l'ensemble des pièces possibles.

- Réaliser au minimum **1 test** de validité **par type** de déplacement
  (ex. Pion avançant en ligne droite, pion avançant en diagonale lorsqu'un adversaire est présent, notez que les pions noirs et blancs n'avancent pas dans la même direction. Tour avançant suivant x>0, cas d'une tour avançant suivant x<0, cas d'une tour avançant verticalement, présence d'une pièce adverse ou non, etc.)

- Pour les tests unitaires et pour les tests de fonctionnalité invalides, réalisez un **unique test** par fonctionnalité.
  (ex. Ne testez pas en même temps la tour et le fou, ne testez pas l'un après l'autre deux coups invalide puisque seul le premier passera à l'exécution, …)

- Réalisez quelques tests **d'intégrations** aboutissant à une configuration de plateau plus complexe.

- Pensez bien aux cas de coups **invalides**, aux **cas particuliers**.
  Etablir les tests des cas particuliers dès maintenant permet de ne pas les oublier plus tard au moment du développement de code.

- Utilisez les **outils** à disposition pour simplifier la mise en scène des tests: *lecture de fichiers de scènes, initialisation, commentaires*.

Format demandé:

Chaque test doit être écrit dans un **fichier séparé**.
On rangera ces fichiers dans une **arborescence spécifique**:

```
test/test_unitaire_ok/test_unitaire_ok_01.txt
                     (test_unitaire_ok_01_entree.txt)
                      test_unitaire_ok_02.txt
                     (test_unitaire_ok_02_entree.txt)
                      test_unitaire_ok_03.txt
                     (test_unitaire_ok_03_entree.txt)
                      ...
    /test_unitaire_ko/test_unitaire_ko_01.txt
                     (test_unitaire_ok_01_entree.txt)
                      test_unitaire_ko_02.txt
                     (test_unitaire_ok_02_entree.txt)
                      test_unitaire_ko_03.txt
                     (test_unitaire_ok_03_entree.txt)
                      ...
    /test_integration/test_integration_01.txt
                      test_integration_01_entree.txt
                      test_integration_01_sortie.txt
                      test_integration_02.txt
                      test_integration_02_entree.txt
                      test_integration_02_sortie.txt
                      test_integration_03.txt
                      test_integration_03_entree.txt
                      test_integration_03_sortie.txt
```

Tests OK:
Les fichiers de **`test_unitaires_ok`** sont des tests de **déplacement valides** qui doivent aboutir à un déplacement d'une pièce.

Ils possèdent la structure donnée en exemple.

```
#test_unitaire_ok_01
#
#NOM1: ZZZZ
#PRENOM1: ZZZZ
#NOM2: ZZZZ
#PRENOM2: ZZZZ
#GROUPE: ZZ
#
# Ce test permet de verifier qu'un pion puisse se deplacer en ligne droite d'une
case lorsqu'il n'y a pas d'obstacles.
#
i
d 1 1 -> 1 2
fin
#Sortie attendue:
# Pion initialement place en (1,1) est deplace en (1,2)
```

Le test précise en commentaire:

- le nom des auteurs,
- le but du test,
- les lignes de contrôles effectivement envoyées,
- la sortie attendue.

➔ Appelez votre programme avec la syntaxe suivante, testez avec le fichier exécutable solution:
`./projet_3eti_solution < test/test_unitaire_ok/test_unitaire_ok_01.txt`

➔ Quelle est la configuration de sortie après exécution de ce test?

Note: Le symbole “<” indique la **redirection d'entrée standard**. C'est à dire qu'à la place de venir lire les instructions sur la ligne de commande, le programme va les lire dans le fichier.

Pour directement s'aider d'une configuration particulière, on pourra écrire en dur des **fichiers** de configuration d'échiquier (voir documents annexes).

On pourra prendre exemple sur le cas *03* donné dans le jeu de tests fourni.
Notez que pour vous aider, vous pouvez visualiser l’échiquier correspondant à ce fichier en appelant le visualiseur de cette manière:
`./visualiseur ../test/test_unitaire_ok/test_unitaire_ok_03_entree.txt &`

Pour aller plus loin.
Lorsque beaucoup de fichiers de tests sont ecrits, il est utile de pouvoir les lancer automatiquement.
Ecrivez le fichier lanceur.sh suivant:

```
================
#!/bin/bash

./projet_3eti_solution < test/test_unitaire_ok/test_unitaire_ok_01.txt
./projet_3eti_solution < test/test_unitaire_ok/test_unitaire_ok_02.txt
./projet_3eti_solution < test/test_unitaire_ok/test_unitaire_ok_03.txt
================
```

Puis tappez en ligne de commande:
$ chmod +x lanceur.sh
$ ./lanceur.sh

Observez que les tests sont exécutés les uns derrière les autres.

Il est également possible de sauvegarder la sortie du projet dans un fichier plutôt que de l'avoir à l'écran. On se sert alors de la redirection de sortie. On pourra par exemple lancer:
$ ./projet_3eti_solution < test/test_unitaire_ok/test_unitaire_ok_01.txt > mon_fichier_de_sortie.txt

Observez alors le contenu de mon_fichier_de_sortie.txt

Tests KO:

Les tests du répertoire **`test_unitaire_ko`** doivent tester des déplacements ou des entrées textuelles invalides.
Testez bien chaque **cas particulier** que vous devrez gérer au niveau du code.

Lorsque vous coderez les fonctionnalités, vous devrez retrouver au moins un fichier de test correspondant à chaque cas particulier du code.
Vous pouvez vous inspirer des fichiers d'exemples fournis.

Pour aider à coder vos fonctionnalités, vous commenterez également le **type de message d'erreur attendu**, ainsi qu'à quel **niveau** cette erreur devra être détectée (parseur, API, déplacement propre à une pièce, etc.)

Tests d'intégrations:

Les tests d'intégrations vérifient des **combinaisons** de déplacements valides plus complexes afin de permettre de vérifier que le système global s'articule correctement.

La différence avec le `test_unitaire_ok` provient du fait que l'on ne vise pas à tester une unique fonctionnalité mais le **comportement global**.

Les tests aboutissent à des configurations plus complexes, et donc longues à vérifier. Il est avantageux de définir en supplément l'état formel de l'échiquier attendu après le déplacement des pièces (`fichier test_integration_XX_sortie.txt`).
De cette manière, il est possible de **comparer** les fichiers du résultat obtenu avec celui du résultat attendu afin de s'assurer que le résultat est bien celui escompté.

On pourra s'inspirer des exemples fournis dans le répertoire des tests d'intégration.

Lors le programme sera complété, vous pourrez exécutée les tests d'intégrations avec la syntaxe:
```
$ ./projet_3eti_solution < ../test/test_integration/test_integration_01.txt
```

➔ Comparez le fichier d'état attendu par rapport à l'état obtenu visuellement.

Pour aller plus loin:

Ecrivez le fichier le fichier d'état normalement obtenu dans echiquier_courant.txt
Comparez le fichier d'état attendu par rapport à l'état obtenu à l'aide de la commande diff:
$ diff -i -E -b -w -B echiquier_courant.txt ../test/test_integration/test_integration_01_sortie.txt

Modifiez le fichier test_integration_01_sortie.txt en modifiant les coordonnées d'une pièce.
Relancer la commande diff, et observez le résultat.
Concluez sur la mise en place d'une procédure de test automatique.

➔ Implémentez les différents tests.

Ecriture de l'état de l'échiquier.

(durée max: 2h)

L'échiquier est une structure contenant 2 jeux et un entier:

- le jeu des pièces blanches
- le jeu des pièces noires.
- le joueur courant (entier valant 0:blanc ou 1:noir).

L'état de l'échiquier peut être initialisé à l'aide de la fonction `echiquier_initialise`.

Pour récupérer le type et les coordonnées de la 3ème pièce du jeu noir, on pourra utiliser la syntaxe:

```
int main()
{
    echiquier echiquier_courant;
    echiquier_initialise(&echiquier_courant);
    jeu *jeu_courant=&(echiquier_courant.jeu_de_piece_noir);
    piece* piece_numero_3=&(jeu_courant->ensemble_de_piece[3]);
    type type_courant=piece_numero_3->type;
    int x_courant=piece_numero_3->coord_x;
    int y_courant=piece_numero_3->coord_y;

    //de la même maniere, on peut acceder à un champ précis sans étapes
intermédiaires:
    type type_courant2=echiquier_courant.jeu_de_piece_noir[3].type;
    int x_courant2 = echiquier_courant.jeu_de_piece_noir[3].coord_x;
    int y_courant2 = echiquier_courant.jeu_de_piece_noir[3].coord_y;

}
```

L'état complet de l'échiquier peut être **sauvegardé dans un fichier** comme il vous est proposé en exemple.

L'implémentation du corps de la fonction d'écriture vous est donnée en tant que **librairie** (Observez la syntaxe **-l** dans le programme de compilation).

Il s'agit d'un binaire obtenu après compilation, et donc **non lisible** par un humain. Seule l'en-tête vous est donnée.
Il s'agit d'un format d'échange standard de fonctionnalités.
Sous Linux, les librairies possèdent l'extension **.so** (librairie dynamique) ou **.a** (librairie statique).
Les librairies standards sont généralement situées dans le repertoire `/usr/lib/`
Sous *Windows*, les librairies possèdent l'extension **.dll** et sont généralement placées directement à côté de l'exécutable.

Vous devez re-coder la fonctionnalité de cette librairie.
Pour cela, réécrivez le corps de la fonction **`entree_sortie_ecrit_echiquier_fichier`** dans `entree_sortie.c`, et enlevez la ligne de "*link*" à la librairie fournie dans le Makefile (c'est à dire la partie associée à l'option -l).

Pour vous aider à vérifier le contrat, vous disposer d'une fonction vérifiant qu'un fichier est bien accessible.
Pour ouvrir un fichier et écrire en *ASCII*, on peut suivre ce type de syntaxe:

```
int main()
{
    FILE *fid=NULL; //struct contenant un descripteur de fichier
    fid=fopen(filename,"w"); //ouverture du fichier "filename" en mode ecriture

    if(fid==NULL)
    {
        printf("Erreur ouverture du fichier %s\n",filename); exit(1);
    }

    fprintf(fid,"J'ecris dans un fichier le nombre %d \n",3);
    fprintf(fid,"Et je sais que 2+2=%d",2+2);
    fprintf(fid,"Enfin si j'ai une piece de type tour, j'ecrirai le nombre
%d\n",tour);

    int c=fclose(fid);
    if(c!=0)
    {printf("Erreur fermeture fichier %s\n",filename);exit(1);}
}
```

Note: La syntaxe de `fprintf` est équivalente à celle de `printf` à l'exception que le descripteur du fichier est passé en paramètre.

Pour aller plus loin:
La syntaxe:
fprintf(stdout,"J'ecris dans le fichier 'ecran'\n");
est identique à l'utilisation de printf, car stdout pointe vers la sortie de la ligne de commande.

- ➔ **Implémentez** l'écriture de l'état de l'échiquier dans un fichier.
- ➔ **Testez** votre code sur différents états de l'échiquier.

Travail en autonomie

(durée estimé: 3h):

- Révision fichiers, chaîne de caractères (1h).
- Finalisez les scripts de tests et déposez votre résultat (2h)

➔ **Déposez** votre projet (code source+scripts de tests) tel que précisé dans les consignes.

Seance 4:

Interface de contrôle de haut niveau: l'API.

(durée max: 2h)

Nous allons désormais implémenter les méthodes de haut niveau du projet.

➔ **Observez** le fichier `api_echec.c` et `api_echec.h`.

Implémentation de l'API générale:

La désignation d'**API** indique que ce sont les fonctions de ce fichier qui vont servir à interfacer le moteur du déplacement de pièce avec tout autre programme ou librairie.
En d'autre termes, vue de l'extérieur, un programmeur n'a besoin d'accéder qu'aux fonctions décrites dans l'**API** pour réaliser un jeu d'échec jouable.

Les fonction d'écriture et de lecture on dejà été implémentées.
La fonction `api_echec_deplacement_piece` est la fonction de gestion du déplacement des pièces et doit être écrite.

➔ **Observez l'algorithme** que doit satisfaire cette fonction.

➔ **Complétez** cette fonction.

Pour cela, vous utiliserez les fonctions déjà implémentées de `echiquier`. Vous utiliserez également les fonctions prévues de `echiquier_deplacement`, ainsi que la structure stockant les paramètres d'une erreur éventuelle.
Celles-ci sont encore à implémenter, cependant elle permettent d'établir la **structure de haut niveau** du fonctionnement du programme.

Notez que vous gérerez les **deux joueurs** dans cette fonction, les jeux à étudier dépendent donc du **joueur courant**.

Notez que dans cette fonction, vous ne devez pas manipuler de structures de plus bas niveau tel que le *type de pièce*, ou le *tableau* stockant les pièces.
Votre code doit ici être écrit de manière **indépendante du type de stockage** choisi. Cela permet de potentiellement modifier ce stockage de bas niveau dans le futur sans avoir à modifier votre code de gestion du jeu.

Si un déplacement est **valide**, l'affichage en ligne de commande sera le suivant:
`J.n deplace (x0,y0) => (x1,y1)`

avec:
n=le numéro du joueur (0 ou 1)
(x0,y0)=coordonnées de départ
(x1,y1)=coordonnées d'arrivées

Si les **coordonnées de départ** sont **invalides**, on affichera:
`Coordonnees de depart (%d,%d) invalides`

Si les **coordonnées d'arrivées** sont **invalides**, on affichera:
`Coordonnees d'arrivees (%d,%d) invalides`

Enfin, si il n'existe pas de pièces du joueur courant aux coordonnées de départ choisies, on affichera:
`Pas de piece courante aux coordonnees de depart`

Gestion du déplacement:

➔ **Observez** les deux fonctions des fichiers `echiquier_deplacement`.

`echiquier_deplacement_est_valide`, est la fonction permettant de définir s'il est possible de réaliser le déplacement entre deux coordonnées.

Cette fonction en elle même doit dépendre du **type de pièce**, cependant il est avantageux de cacher cette complexité en proposant une **interface unique** de déplacement. C'est le rôle de cette fonction. En d'autres terme, la fonction sert de *façade* pour rediriger l'appel vers la fonction de vérification qui dépend de la pièce actuelle.
Nous implémenterons cette fonction plus tard, pour l'instant, laissez le code à "return 1".

La fonction de **`echiquier_deplacement_realise`**, permet d'effectuer réellement le déplacement d'une pièce.
Notez que cette fois le pointeur vers l'échiquier courant n'est **pas constant**, car les coordonnées d'au moins une pièce vont devoir être **modifiées**.

➔ **Implémentez** la fonction de déplacement effectif d'une pièce vérifiant les caractéristiques décrites.

➔ **Testez** votre programme. Celui-ci doit désormais etre fonctionnel et pouvoir être interfacé avec le visionneur.

➔ **Testez** certains de vos **fichiers de tests**, et notez que les erreurs de déplacement propres à un type de pièce ne sont pas pris en compte. Par contre, les autres comportements (dépassement des limites du jeu, existence de pièce à l'arrivée, …) doivent quant à eux être correctement pris en compte.

➔ **Implémentez** désormais la fonction **`echiquier_deplacement_est_valide.`** Vérifiez les **pré-requis**. Puis commencez à implémentez les fonctions de déplacement propre à chaque pièce.

Commencons par implémenter le **déplacement du cavalier**.
Pour cela, on implémentera la fonction:

```
int echiquier_deplacement_est_valide_cavalier    (const echiquier*
echiquier_courant,const piece* piece_a_deplacer,int
x_destination,int y_destination,code_erreur_deplacement *erreur);
```

Cette fonction, **spécifique** d'une pièce donnée, ne doit **pas être visible** depuis l'API de haut niveau. Pour cela, nous allons définir ces fonction dans des **fichiers distincts**.

➔ **Créez** les fichiers:
`echiquier_deplacement_privee.c`, et
`echiquier_deplacement_privee.h`.

Ces deux fichiers vont contenir les **implémentations privées** du déplacement propre à chaque pièce.

En séparant ces implémentations de l'appel générique standard de "`echiquier_deplacement`", cela permet de:

- Faciliter **l'évolution** du code si l'on doit modifier le comportement d'une pièce.
- Permettre de compiler et diffuser une **librairie** implémentant le déplacement des pièces sans avoir à diffuser le code source du corps de ces fonctions.
- Permet de mieux **séparer** les appels de haut et de bas niveau de votre projet.

Implémentons la fonction de déplacement du cavalier dans ces fichiers.

➔ Quels **contrats** satisfait cette fonction de déplacement? **Complétez** l'en tête avec ce contrat.

➔ **Implémentez** le corps de la fonction.

➔ **Testez** votre fonction en déplaçant des cavaliers dans le jeu. Vérifiez les cas valides, vérifiez également la bonne récupération du type de l'erreur dans les cas de déplacements invalides.

Fonction de déplacement propre:

(durée max: 2h)

➔ **Implémentez** les autres fonctions de déplacements: *pions, tour, cavalier, fou, roi, reine*.

➔ **Testez** les déplacements au fur et à mesure.

Travail en autonomie.

- Révision (1h)
- Rendu des fonctions de déplacement (2h)

➔ **Achevez** les fonctions de déplacement de pièces.

➔ **Déposez** votre projet.

Seance 5:

Makefile:

(durée max: 2h)

Jusqu'à présent le script de compilation vient compiler chaque fichier l'un derrière l'autre de manière systématique. Nous avons pris soin de définir l'ordre dans lequel chaque compilation doit avoir lieu, et nous compilons chaque fichier même si celui-ci n'a pas été modifié.
Ceci entraîne un temps de compilation allongé non compatible avec des projets volumineux.

Nous allons utiliser l'outil **`make`** pour simplifier et enrichir la gestion de ce processus de compilation.

`Make` est un outil standard permettant d'automatiser et d'enchaîner des taches de manière simple.

Note: Make est totalement **indépendant** du compilateur `gcc`, et peut être utilisé pour réaliser d'autres tâches que la compilation.

L'appel à `make` en ligne de commande vient lire un fichier du nom de **Makefile**.
La syntaxe associée dans le fichier Makefile suit le modèle suivant:

```
fichier_a_realiser : dependances
     ligne(s) de commandes a executer
```

Explication:
fichier_a_realiser = Le fichier que l'on cherche à créer (ou nom de l'action à réaliser).
dépendances = Le ou les fichiers qui doivent être présents afin de créer ce fichier
ligne(s) de commandes à exécuter = La commande ou les commandes qui doivent être lancées afin de réaliser ce fichier. Chaque ligne est précédé d'une **tabulation** (et non d'espaces).

➔ Observez et exécutez les différents Makefile d'exemples (répertoire `03_makefile/`).

Vous devez être capable de réaliser vos propres `Makefile` et de mettre en place diverses actions grâce à ceux-ci.

➔ **Realisez** le **`Makefile`** pour le projet actuel.

Votre `Makefile` doit satisfaire aux critères suivants:

- L'appel à
  `$ make`
  compile le projet et génère l'exécutable.
- La compilation doit être en mode **debug**, avec un maximum d'options de **Warning**.
- L'appel à
  `$ make clean`
  efface les fichiers temporaires (fichiers .o, et exécutable).

Pour aller plus loin :
Si vous êtes en avance, répondez au questions du répertoire `03b_makefile_avance/`

Réalisation des tests prévus.

(durée max: 2h)

- ➔ **Réalisez** les tests que vous aviez prévus.
- ➔ **Analysez** les résultats obtenus.
- ➔ Devez vous ajoutez des tests supplémentaires?
  Si oui, placez les dans un répertoire: `tests_supplementaires/`

Travail en autonomie.

- Révision + fin d'écriture du Makefile (1h)
- Ajout de tests supplémentaires (2h)

➔ **Uploadez** votre projet avec votre Makefile et vos tests supplémentaires sur le dépôt de fichiers.

Seance 6:

Mise en place d'une librairie.

(durée max: 2h)

Une **librairie C** est un ensemble formé de deux types de fichiers:

- Un ou plusieurs fichiers d'en tête (.h)
- Un fichier (potentiellement plusieurs) binaire contenant le code compilé implémentant les fonctions décrites dans les en-têtes.

Le fichier binaire peut être vue comme une archive regroupant un ou plusieurs fichiers objets (.o).

Il existe deux types de librairies:

1. **Librairie statique** (extension .a), qui vont être incluse dans l'exécutable finale.
2. **Librairie dynamique** (extension .so), qui ne sont pas incluse dans l'exécutable finale, mais liée dynamiquement à chaque lancement du programme. Ce sont les librairies les plus rependues.

Sous Windows, les librairies possèdent l'extension .dll

Les librairies permettent **d'échanger** aisément des **fonctionnalités**, sans avoir à fournir le code du corps des fonctions. Cela peut avoir deux buts:

- **Cacher** au client le code.
- **Simplifier** la démarche d'installation en évitant que le client ne doive compiler du code.

Un exemple typique d'utilisation concerne les mises-à-jour de programme. Lorsqu'une nouvelle fonctionnalité, ou une correction est mis en place. Son installation revient simplement à remplacer la librairie précédente par la nouvelle (changement de .so / ou de .dll sous Windows). L'exécutable du programme (souvent beaucoup plus volumineux) restant inchangé.

Organisation des librairies sous Linux:

Sous Linux, les librairies sont généralement stockés dans des répertoires standards dans lesquels les programmes viennent les chercher par défauts.

➔ **Observez** les répertoires:

- **`/usr/include/`** : il contient les fichiers d'en tête des librairies installées sur votre système.
- **`/usr/lib/`** : il contient les librairies binaires installées sur votre système.

Toutes les librairies visibles possèdent des fonctionnalités (en C ou C++) que vous pouvez utiliser dans vos programmes.

Note: Ces répertoires de librairies ne sont pas à confondre avec les programmes executables disponible dans **/usr/bin/**

Note: Sous *Windows*, il n'existe pas de répertoire standard pour stocker les librairies. Celles-ci sont généralement dupliquées localement dans chaque dossier d'un programme spécifique. Chaque programme viens alors directement charger sa version locale du fichier dll.

Creation d'une librairie:

Considérez l'exemple_1 des fichiers fournis (répertoire `04_librairie/`).

Le dossier `code_librairie/` contient le code source de la librairie.
Le dossier `executables/` contient 2 programmes faisant appel à cette même librairie.
Nous allons générer les deux exécutables en les liant dynamiquement à la librairie.

Premièrement, nous générons la **librairie dynamique**. Cette étape à lieu **une unique fois** à partir du code source de la librairie.

➔ Placez vous dans le répertoire `code_librairie/`

➔ **Compilez** le code source vers un fichier objet avec l'option "**`fPIC`**" (rend la création de librairie dynamique possible).
```
$ gcc -c vecteur.c -fPIC -Wall -Wextra
```

La librairie dynamique se réalise en appelant `gcc` avec l'option "**`shared`**" sur l'ensemble des fichiers objets que l'on souhaite inclure dans cette librairie.

➔ Dans notre cas, on écrit alors:
```
$ gcc -shared vecteur.o -o libvecteur.so
```

Nous avons créé la librairie dynamique.
Celle-ci peut se lier à d'autres programmes sans avoir à utiliser le code source (.c).
Par contre, le fichier d'en tête .h reste **nécessaire** pour utiliser les fonctions définies dans cette librairie.

➔ **Copiez** désormais le fichier d'en tête, et la librairie dans le répertoire `executables/`

➔ **Compilez** les deux programmes exécutables:
```
$ gcc -c mon_programme_1.c -Wall -Wextra
$ gcc -c mon_programme_2.c -Wall -Wextra
```

Lors de l'**édition de liens**, la librairie doit être **associée** au programme.

Pour cela, on utilise l'option **`-l`**`<NOM_LIBRAIRIE>` pour une librairie contenue dans le fichier `libNOM_LIBRAIRIE.so`

➔ Ecrivez:
`$ gcc mon_programme_1.o -lvecteur -o mon_programme_1`

Cette commande échoue. Elle vous indique que la librairie vecteur n'est pas trouvée. En effet, `gcc` viens chercher les librairies dans le chemin standard. Par exemple **/usr/lib/**
Le repertoire courant n'est pas contre pas dans les chemins de recherche par défaut. Pour ajouter un chemin spécifique non standard, on utilise l'option **`-L<CHEMIN>`**.

Dans notre cas, le chemin spécifique est le répertoire courant.

➔ Ajouter le nom complet, ou simplement un point (.) après la lettre **`L`**.
`$ gcc mon_programme_1.o -lvecteur -L. -o mon_programme_1`

Cette fois l'exécutable est généré, la librairie à été trouvée.

➔ Tentez désormais d'exécuter votre programme. Que ce passe t-il?

La ligne d'erreur indique que la librairie n'est pas trouvée. Encore une fois, lors de l'exécution d'un programme exécutable, les librairies dynamiques sont cherchées dans un chemin par défaut : par exemple **`/usr/lib/`**.

➔ Lancez la commande:
`$ ldd ./mon_programme_1`
qui permet de *visualiser* quelles librairies dynamiques sont utilisées par un executable donnée.
Ici, notez que la librairie "`vecteur`" n'est pas trouvée.

Pour **ajouter** un chemin spécifique de recherche, on peut utiliser une **variable globale du Shell** (ligne de commande) qui contient la liste des répertoire à visiter.
Cette variable porte le nom de **`LD_LIBRARY_PATH`**. Pour ajouter un répertoire `<CHEMIN>`, utilisez la syntaxe suivante en ligne de commande:
`$ export LD_LIBRARY_PATH=<CHEMIN>:$LD_LIBRARY_PATH`

➔ **Appliquez** cette ligne dans le cas où `<CHEMIN>=.` (repertoire courant).
`$ export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH`

➔ **Relancez** votre exécutable.
Cette fois la librairie est trouvée. Elle est cherchée et chargée dynamiquement à chaque lancement.

➔ **Observez** à nouveau le résultat de:
`$ ldd ./mon_programme_1`

➔ Ouvrez un nouveau terminal dans ce même répertoire. Tentez de lancer votre exécutable à partir de ce nouveau terminal. Que ce passe-il?

Cette fois, le chemin d'accès ne contient plus le répertoire courant, et la démarche d'affectation de la variable du Shell est à nouveau nécessaire.

Note: Sous *Windows*, le chemin par défaut est au contraire le répertoire courant. C'est pour cette raison qu'il est courant de devoir déplacer les .dll à coté d'un exécutable.

Pour aller plus loin:
Afin d'éviter cette manipulation à chaque lancement, il est possible de définir la ligne export LD_LIBRARY_PATH avec les chemins spécifique dans un script qui viens ensuite lui même lancer l'exécutable.

➔ **Réalisez** les autres exemples dont l'énoncé est contenu dans les fichiers `consignes.txt.`

➔ **Creez** une librairie dynamique du comportement de déplacement de vos pièces (`piece_deplacement_privee`) sous le nom de `libdeplacement.so.`

➔ **Modifiez** votre Makefile tel que l'appel à
`$ make -f Makefile_librairie`

génère un exécutable du projet liée à libdeplacement.so

Pour aller plus loin :
Si vous êtes en avance, répondez aux questions contenues dans le répertoire `04b_librairie_avance/`

Rendu d'un logiciel livrable.

(durée max: 2h)

➔ **Finalisez votre projet.**

➔ **Ajoutez la condition de mise en échec et de victoire.**

Note:

- *Avez-vous suivit les règles de bonne programmation?*
- *Assurez vous que l'ensemble de vos fonctions sont correctement commentés.*
- *Assurez vous de respecter une programmation par contrats. Les fichiers .h sont ils commentés, les corps des fonctions respectent-elles les contrats?*
- *Gérez-vous correctement le mot clé const?*
- *L'ensemble des tests-ont ils été réalisés ? Tous les cas d'erreurs sont-ils testés?*
- *Votre* `Makefile` *est-il à jour, est-il lisible? Peut-on factoriser des arguments?*

➔ Synthétisez les tests effectués et expliquez votre démarche ainsi que vos choix dans un **compte-rendu** de 5 à 10 pages.

Travail en autonomie:

- Rendu du projet (3h)

➔ **Archivez** votre projet.
Vous devez rendre l'ensemble de votre **code source**.
Un fichier **binaire** et sa **librairie** devra être rendu dans un répertoire spécifique.
Votre **compte-rendu** au format pdf doit être contenu dans un répertoire spécifique (`compte_rendu/`).

➔ **Déposez** votre projet final sur le dépôt de fichier.

Bonus:

Si vous êtes arrivés à ce point avant la fin des 6 séances, et que vous avez reçu la validation des enseignants.

- ➔ **Réalisez** l'implémentation d'un joueur adverse autonome en mettant en place une **intelligence artificielle**.

Votre implémentation d'intelligence artificielle devra se lancer à l'aide d'un **appel unique**:
`ai_joue_tour(echiquier* echiquier_courant);`
qui jouera un tour étant donné l'échiquier courant de manière automatique.

Cette fonction (et tout autre structure lui étant liée) sera compilée et accessible sous la forme d'une **librairie dynamique** du nom de `libechec_ai.so`

Si plusieurs librairies sont proposées, un **tournoi** de l'intelligence artificielle la plus performante pourra être mis en place.